

Arquitecturas Basadas en Servicios (Microservicios)

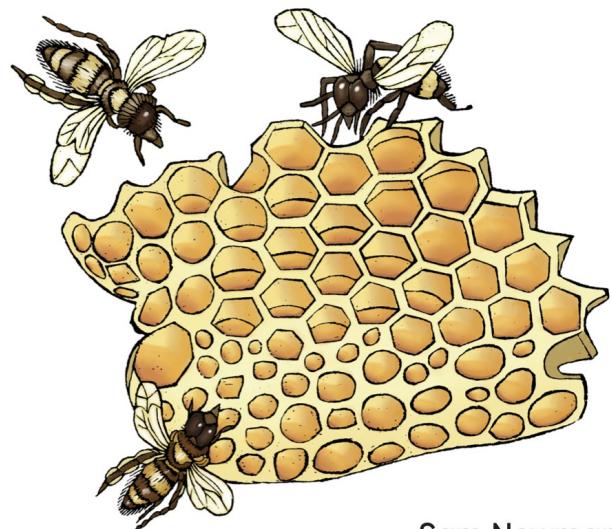
“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

— Grady Booch

O'REILLY®

Building Microservices

Designing Fine-Grained Systems



Sam Newman

Second
Edition

Microservices Patterns

Chris Richardson

MANNING

With examples in Java



Spring Microservices IN ACTION

SECOND EDITION

John Carnell
Hillary Huaylupo Sánchez

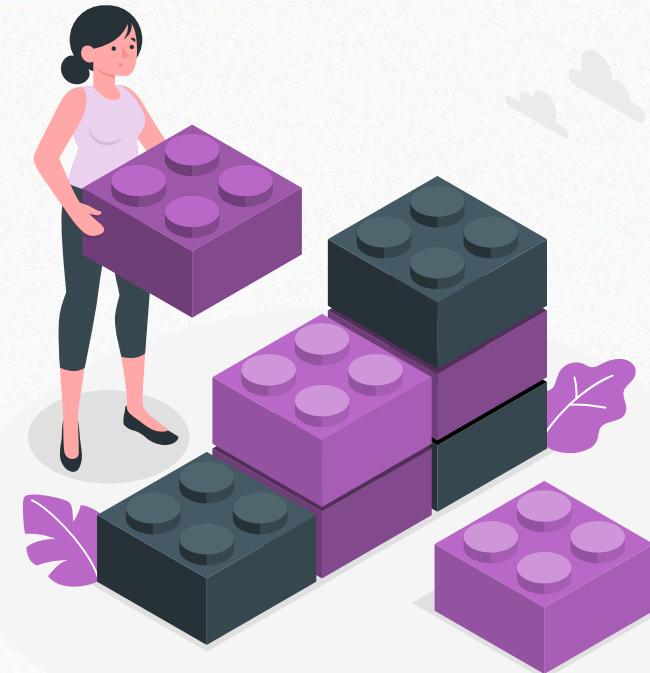
MANNING



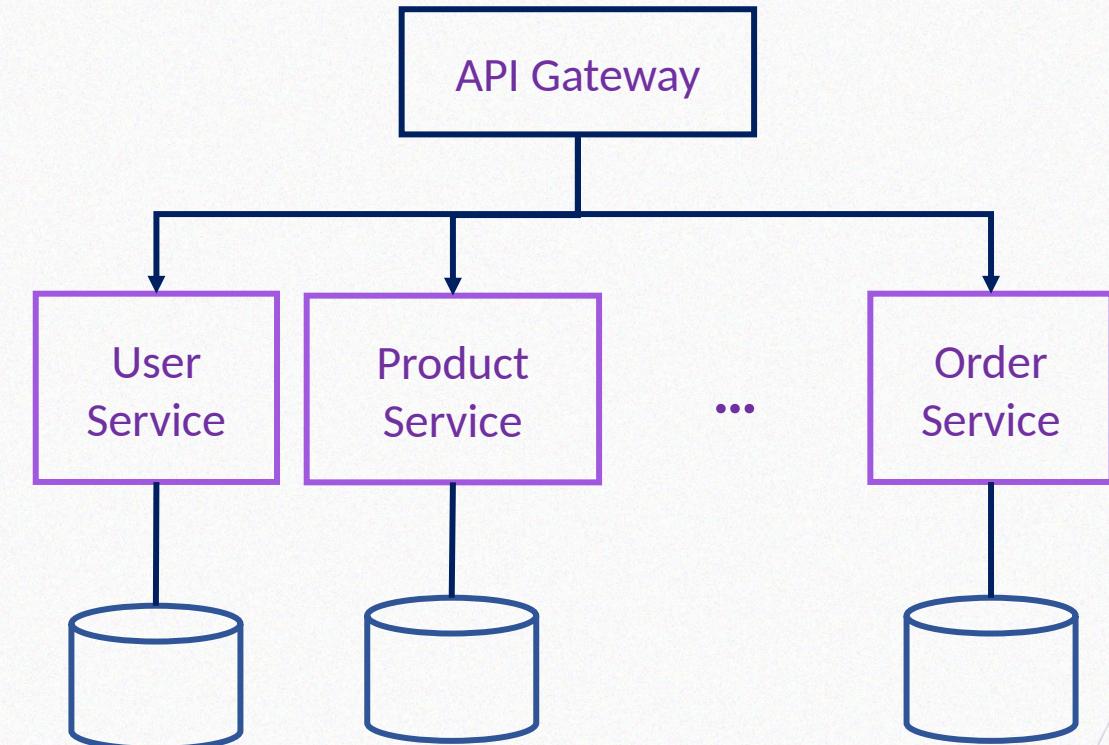
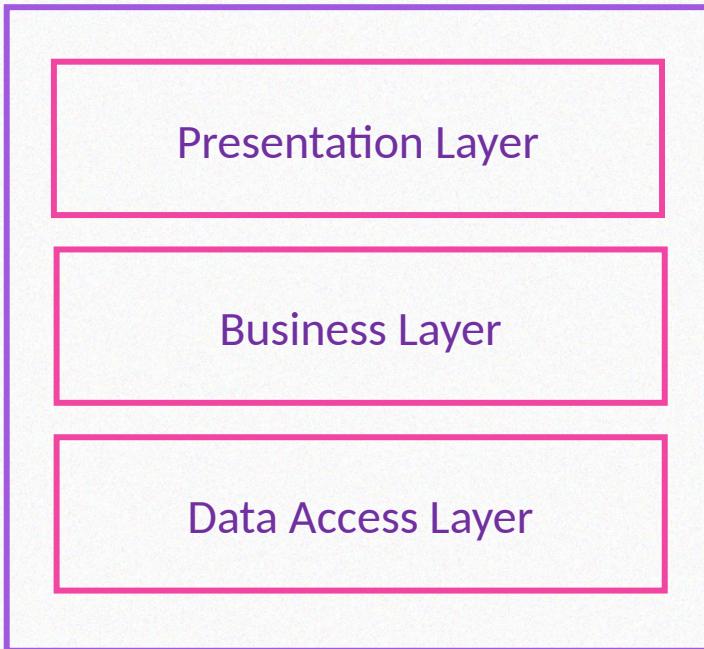
Diferencias entre Monolitos y Microservicios

Monolito: Una arquitectura monolítica es aquella en la que toda la funcionalidad de una aplicación está contenida en una única unidad desplegable.

Microservicios: Los microservicios son un estilo arquitectónico donde la aplicación se descompone en pequeños servicios independientes, cada uno ejecutando una funcionalidad específica del negocio.



Diferencias entre Monolitos y Microservicios



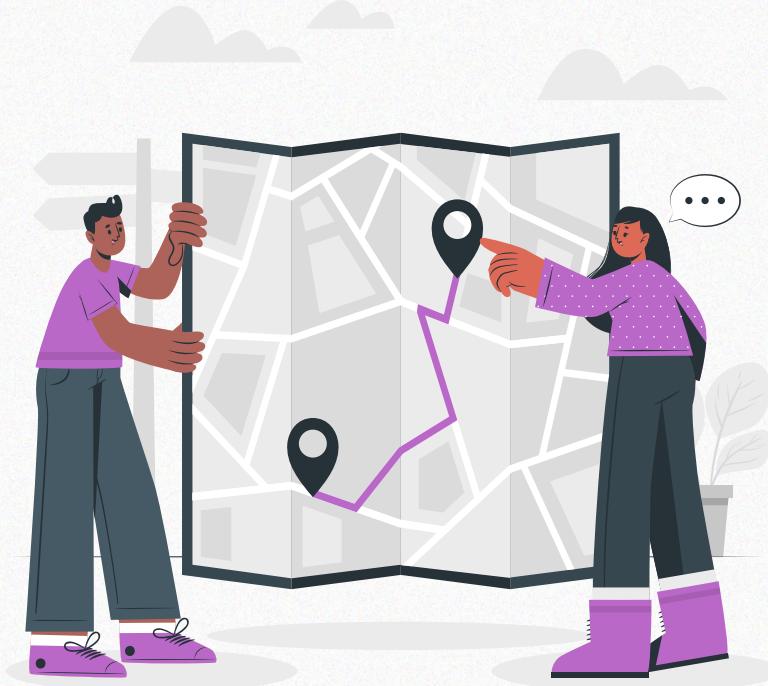
Aspecto	Monolito	Microservicios
Arquitectura	Una sola unidad desplegable.	Múltiples servicios independientes.
Tamaño del Equipo	Ideal para equipos pequeños.	Compatible con equipos distribuidos.
Escalabilidad	Escalabilidad horizontal limitada.	Escalabilidad independiente por servicio.
Mantenimiento	Cambios afectan todo el sistema.	Cambios limitados al servicio afectado.
Despliegue	Todo el sistema debe redeployarse.	Servicios se despliegan individualmente.
Desempeño	Mejor desempeño inicial por comunicación interna.	Penalización por comunicación a través de red.
Gestión de Datos	Base de datos única.	Base de datos independiente por servicio.
Complejidad	Menos compleja inicialmente.	Complejidad alta en gestión de servicios.

Diseño de Microservicios

Patrón Bounded Context

Forma parte del **Domain-Driven Design**
propuesto por Eric Evans

Piensa en un sistema como un mapa político,
donde cada país tiene sus propias leyes, cultura
e idioma, y la frontera define qué aplica dentro de
ese país.



Ejemplo Practico:

Contextos Identificados:

- **Ventas**: Gestiona órdenes, catálogos, precios.
- **Inventario**: Monitorea stock, reposiciones.
- **Logística**: Maneja envíos, rutas y tiempos de entrega.

Problema

“Producto” significa cosas diferentes:

- En ventas, se refiere a descripciones y precios.
- En inventario, al SKU y cantidad disponible.



Ejemplo Practico:

Solución

Definir límites claros:

- **En el contexto de Ventas:** Product tiene atributos name, price.
- **En el contexto de Inventario:** Product tiene atributos sku, stockLevel.



Contexto Ventas

```
class Product:  
    def __init__(self, name: str, price: float):  
        self.name = name  
        self.price = price  
        self.sku = sku
```

Contexto inventario

```
class Product:  
    def __init__(self, sku: str, stock_level: int):  
        self.sku = sku  
        self.stock_level = stock_level
```



Comunicación entre Contextos

Alguien compra un producto en Ventas → inventario recibe un evento para reducir el stockLevel.

```
class ProductPurchasedEvent:  
    def __init__(self, sku: str, quantity: int):  
        self.sku = sku  
        self.quantity = quantity
```

Subscriber en inventario:

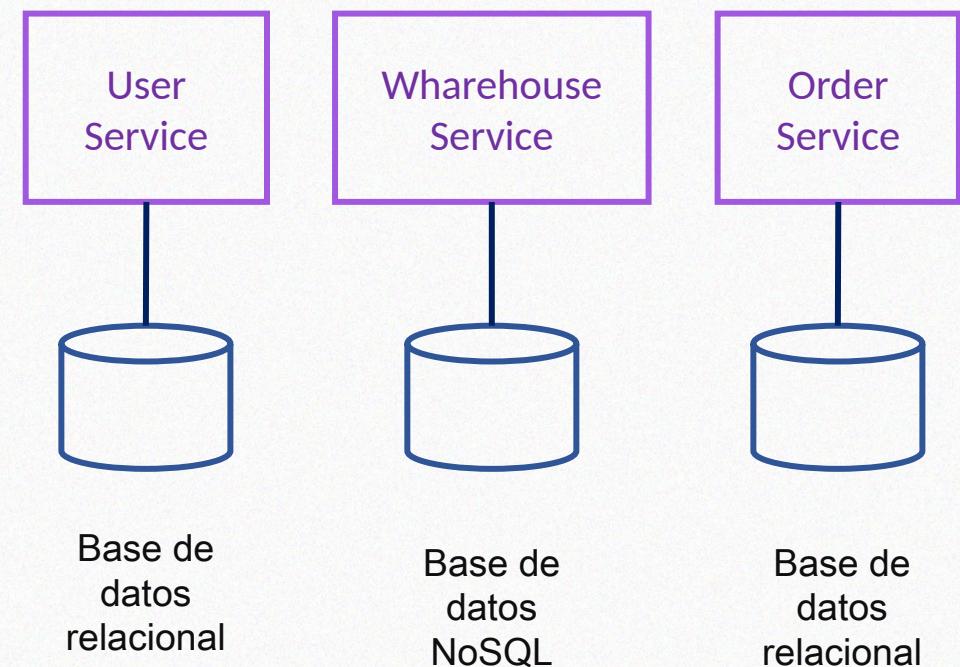
```
def handle_product_purchased(event: ProductPurchasedEvent):  
    inventory_product = get_product_by_sku(event.sku)  
    inventory_product.stock_level -= event.quantity
```



Patrón Database per Service

El patrón “Database per Service” sugiere que cada servicio en una arquitectura de microservicios tenga su propia base de datos, evitando compartir una base de datos monolítica entre múltiples servicios.

- **Autonomía:** Cada servicio es responsable de su propio modelo de datos.
- **Aislamiento:** Los servicios no comparten esquemas ni tablas.
- **Independencia tecnológica:** Cada servicio puede usar la base de datos que mejor se ajuste a sus necesidades (SQL, NoSQL, etc.).



Ventajas

1. Escalabilidad independiente:

- Cada servicio puede escalar horizontalmente sin afectar a otros servicios.

2. Resiliencia:

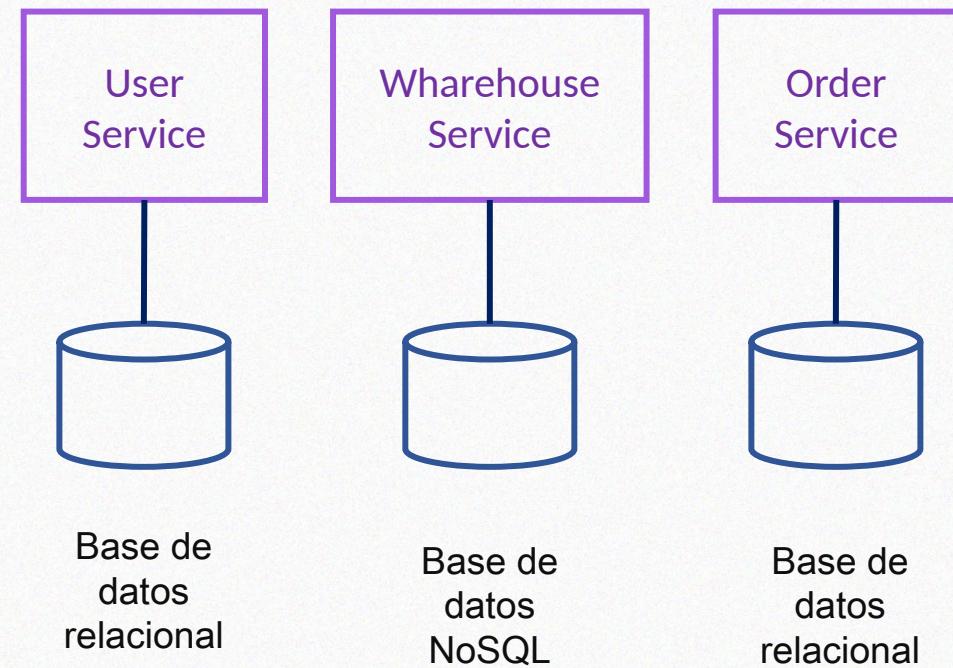
- Un fallo en la base de datos de un servicio no afecta directamente a otros servicios.

3. Flexibilidad tecnológica:

- Cada servicio puede usar la tecnología más adecuada a sus necesidades.

4. Mejora en la mantenibilidad:

- Cambios en el esquema de datos de un servicio no afectan a otros.



Desafíos

1. Complejidad de consultas cruzadas:

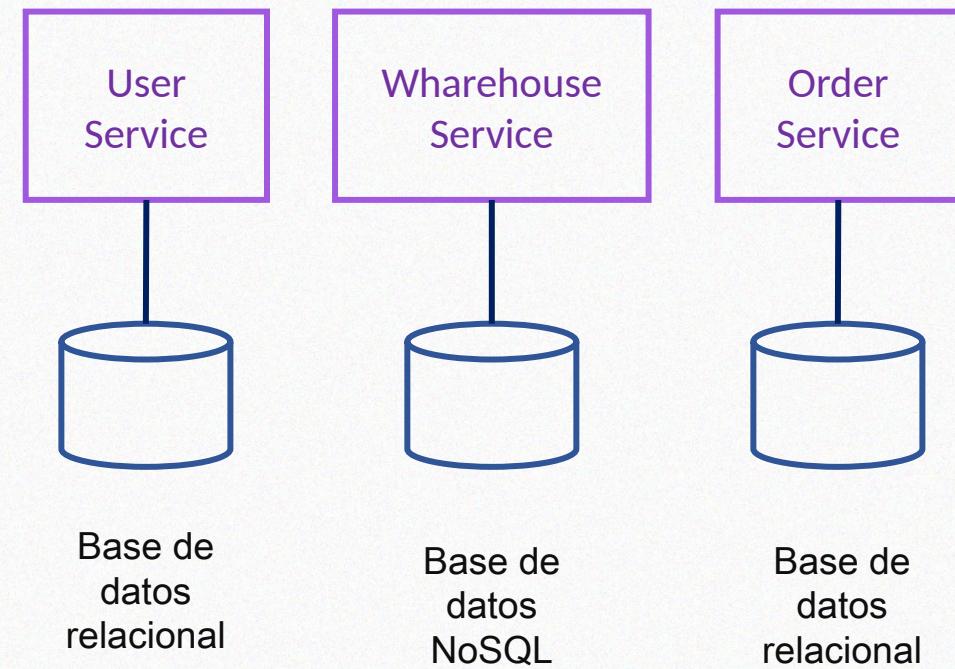
- No se pueden realizar consultas SQL directas entre servicios.
- Se necesita replicar datos o usar eventos.

2. Gestión de transacciones distribuidas:

- Operaciones que afectan a múltiples servicios requieren patrones como **sagas** o **eventual consistency**.

3. Sobrecarga operativa:

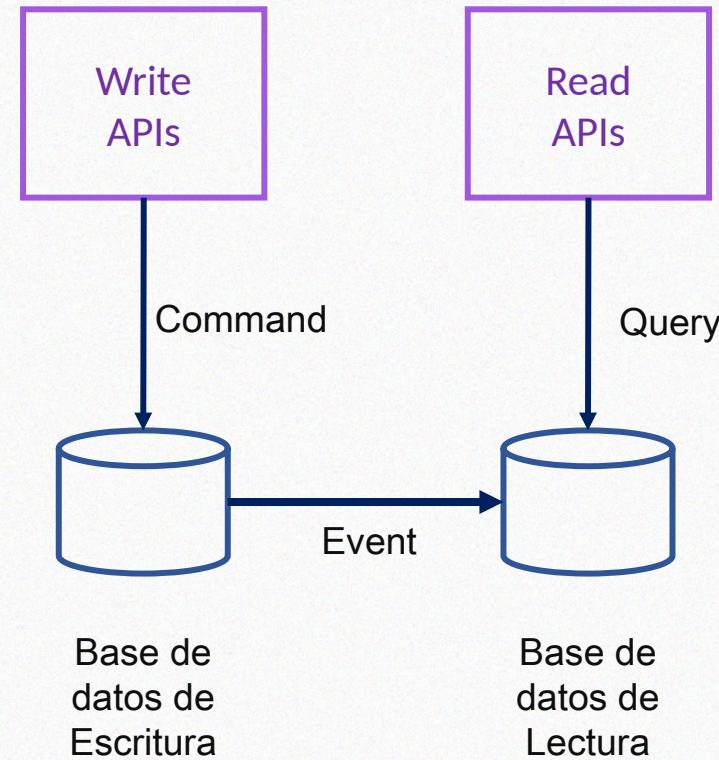
- Mantenimiento de múltiples bases de datos.



CQRS

El patrón **CQRS** separa las operaciones de lectura (**Query**) de las operaciones de escritura (**Command**) en una aplicación, asignándolas a modelos independientes.

“Lecturas y escrituras tienen propósitos distintos y pueden optimizarse por separado”.



Ventajas:

1. Separación de preocupaciones:

- Los modelos de lectura y escritura evolucionan de forma independiente.

2. Escalabilidad:

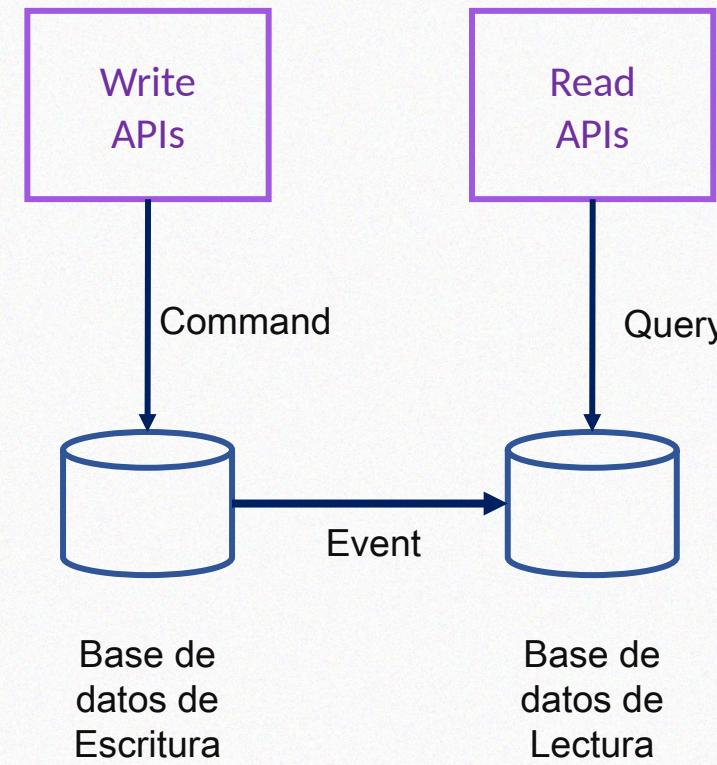
- Los modelos de consulta pueden escalar horizontalmente para manejar cargas pesadas de lectura.

3. Optimización:

- Cada modelo se puede optimizar para su propósito (consultas rápidas o transacciones robustas).

4. Simplificación de la lógica:

- Menos complejidad en la implementación de operaciones.



Desventajas:

1. Mayor complejidad:

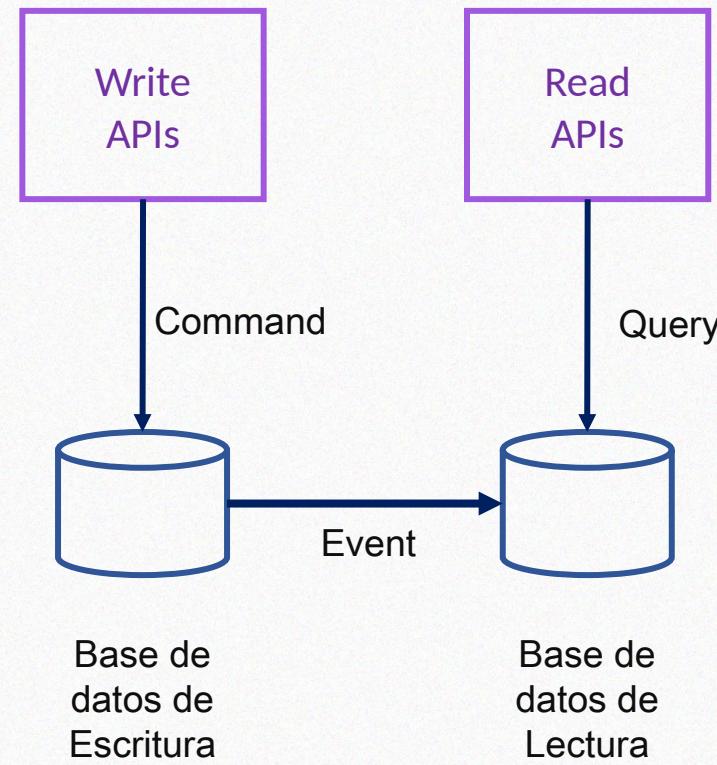
- Introduce sincronización entre modelos de lectura y escritura.

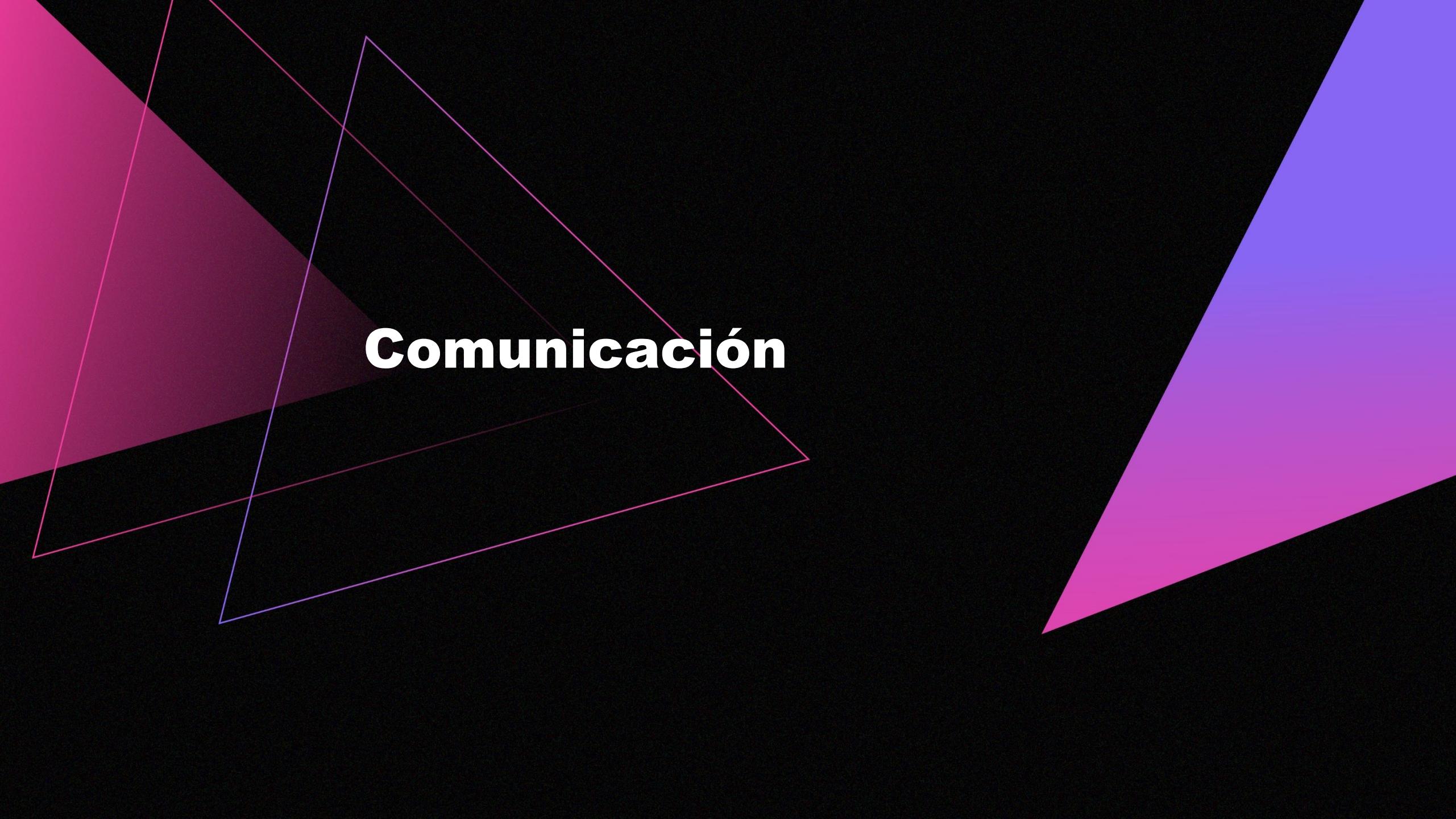
2. Consistencia eventual:

- Puede haber retrasos entre una actualización y la disponibilidad de los datos para consulta.

3. Sobrecarga operativa:

- Mantenimiento de dos modelos separados.

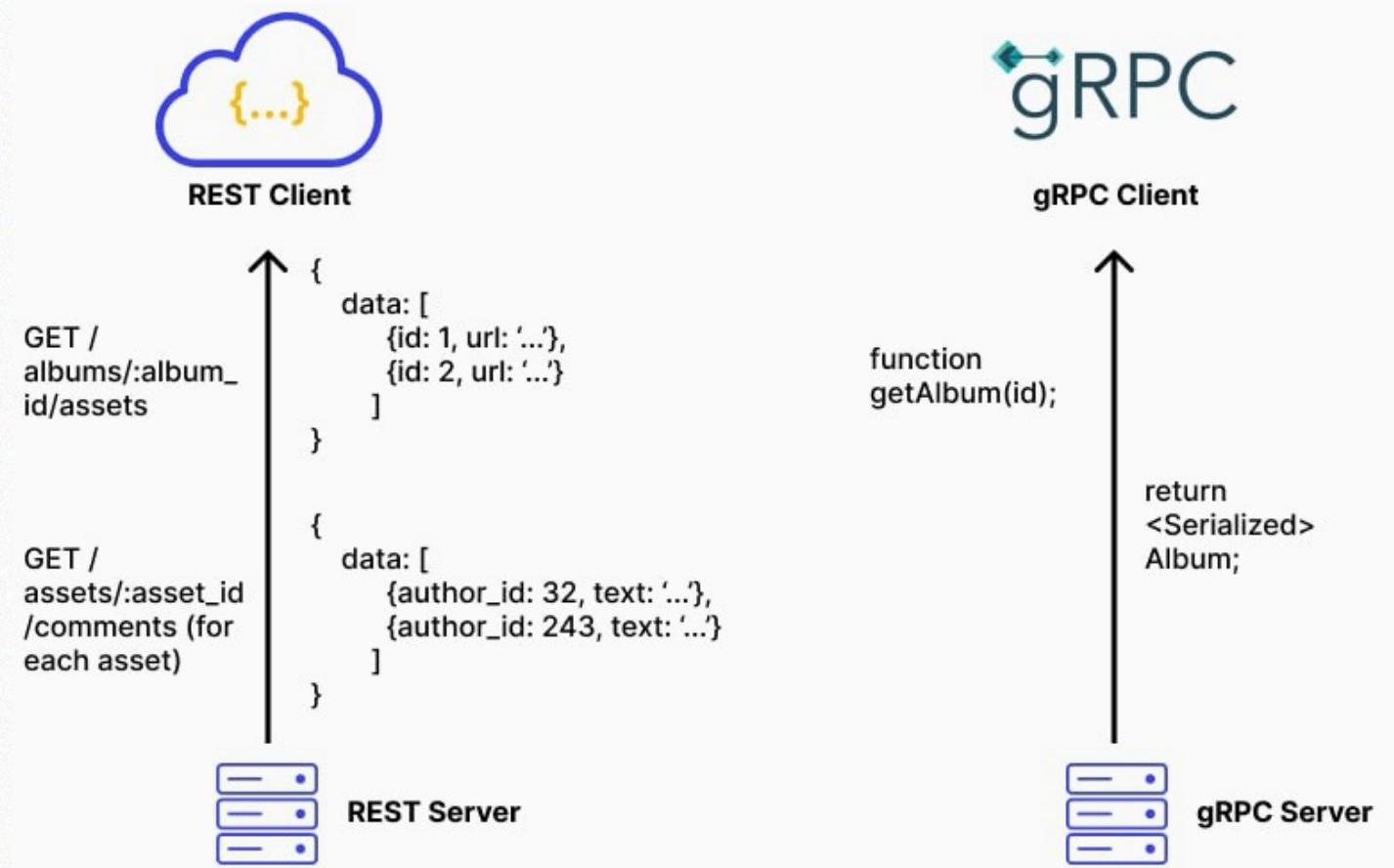


The background features a dark gray gradient with several geometric shapes. In the upper left, there is a large triangle filled with a pink-to-purple gradient. A smaller, thin-lined triangle is positioned above and to the right of it. To the right of the center, a large triangle is filled with a purple-to-pink gradient, with its peak pointing towards the bottom right corner.

Comunicación

Comunicación

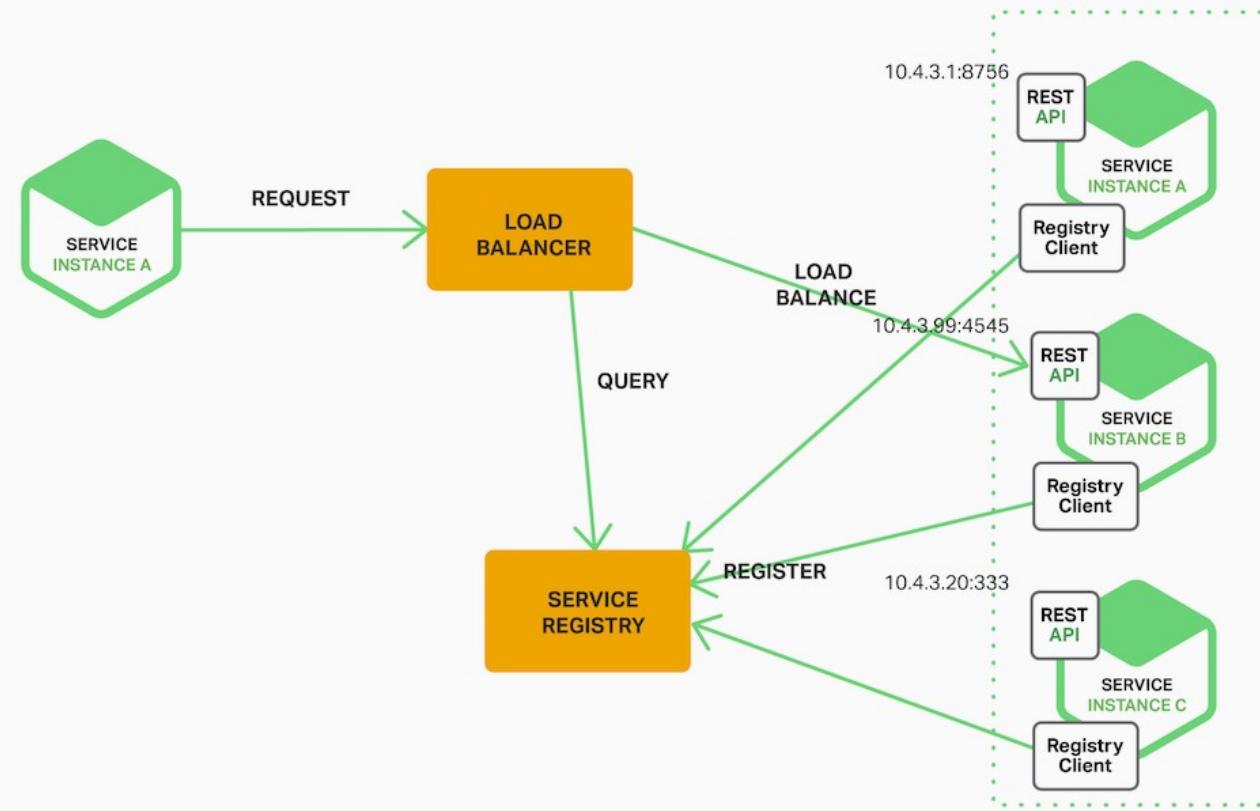
Comunicación Síncronica



Comunicación

Server Registry

El **Server Registry** (o **Service Registry**) es un componente esencial en arquitecturas distribuidas, especialmente en sistemas de microservicios. Se utiliza para gestionar la **descubridibilidad de servicios**, permitiendo que los servicios en el sistema puedan encontrarse y comunicarse de manera dinámica.



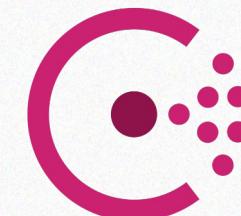
Comunicación

Server Registry

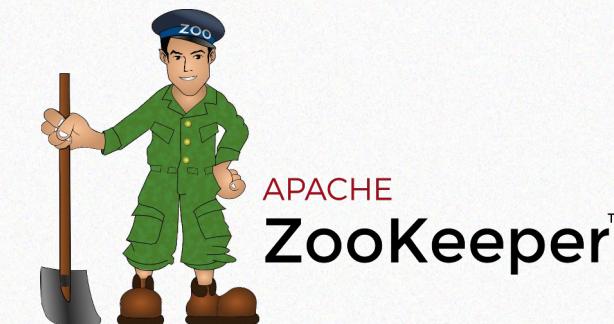
El **Server Registry** (o **Service Registry**) es un componente esencial en arquitecturas distribuidas, especialmente en sistemas de microservicios. Se utiliza para gestionar la **descubridorabilidad de servicios**, permitiendo que los servicios en el sistema puedan encontrarse y comunicarse de manera dinámica.



Eureka



HashiCorp
Consul



Comunicación

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
```

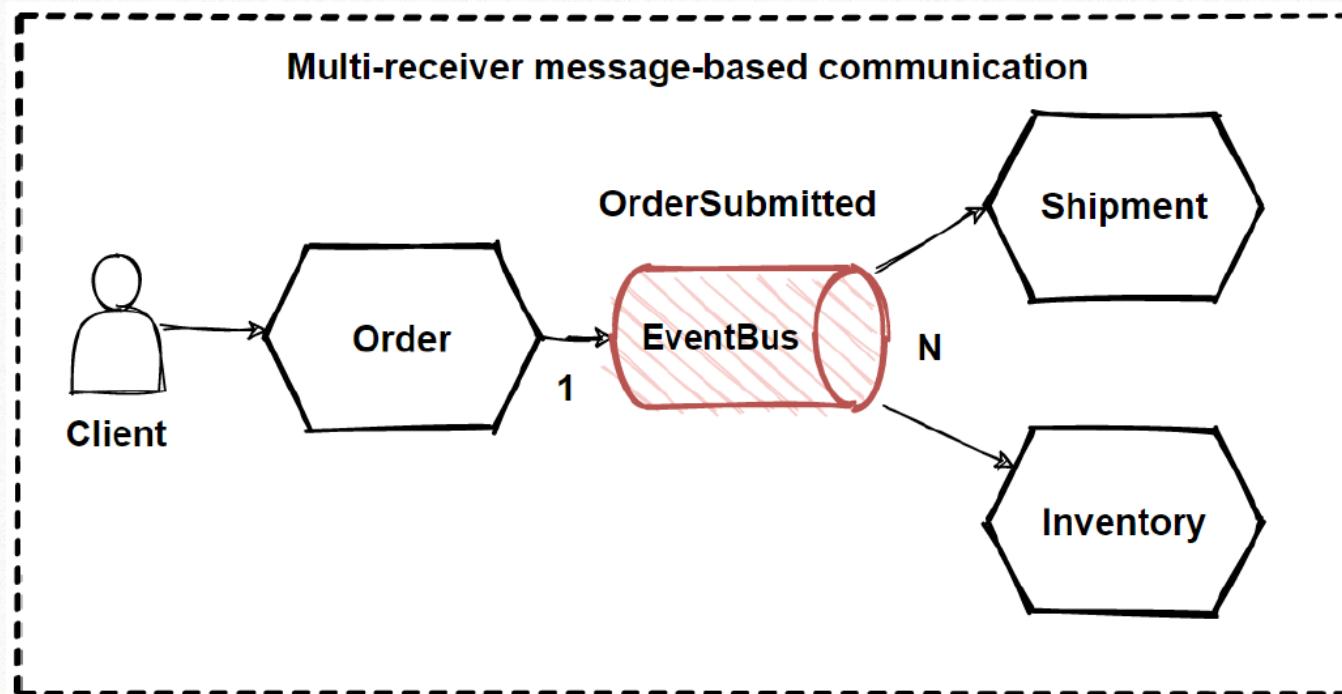
Comunicación

```
@RestController
@RequestMapping("/consume")
public class ConsumerController {
    @Autowired
    private DiscoveryClient discoveryClient;

    @GetMapping("/service-instances/{serviceName}")
    public List<ServiceInstance> serviceInstances(@PathVariable String
    serviceName) {
        return discoveryClient.getInstances(serviceName);
    }
}
```

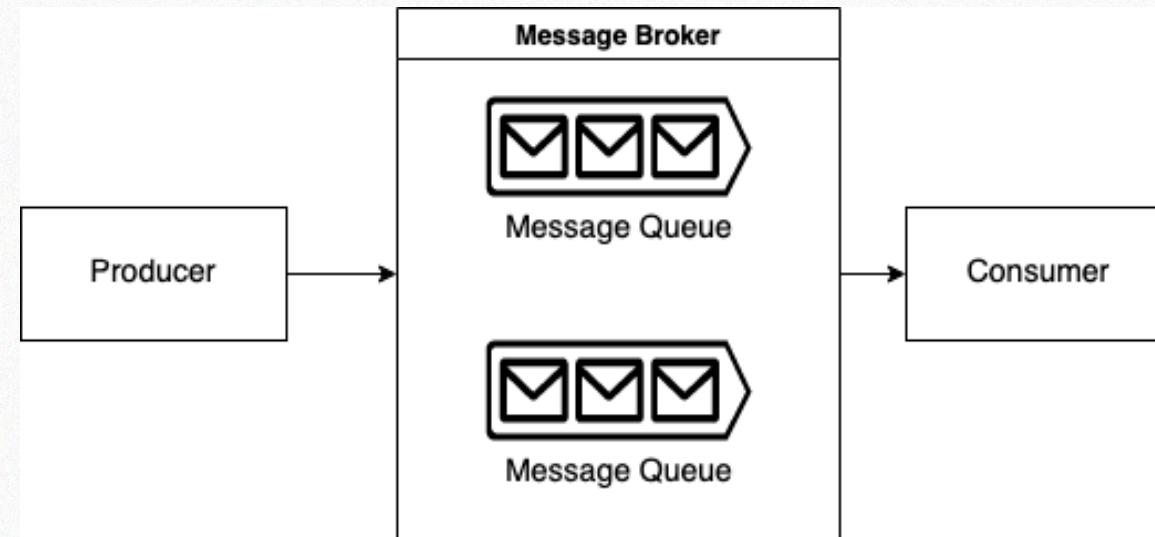
Comunicación

Comunicación
Asincrona



Message Broker

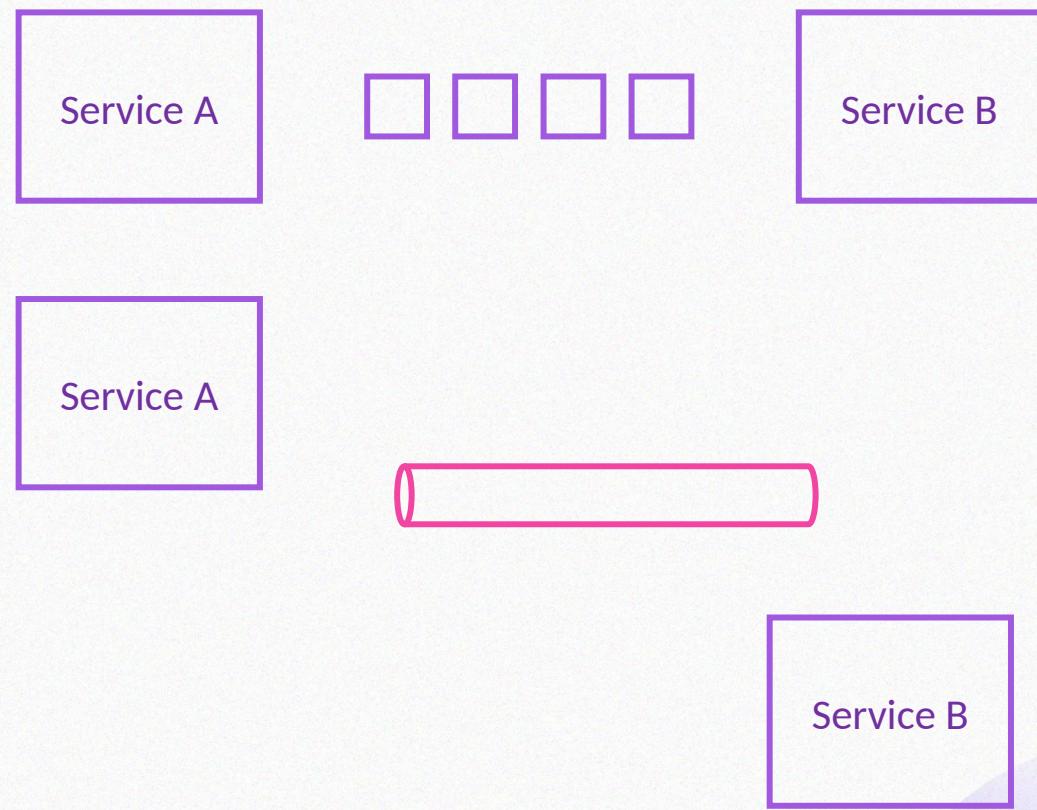
Un **Message Broker** es un intermediario de mensajes que facilita la comunicación entre diferentes servicios en una arquitectura distribuida. En sistemas de microservicios, los Message Brokers juegan un papel crucial para desacoplar los servicios y manejar el flujo de datos de manera eficiente.



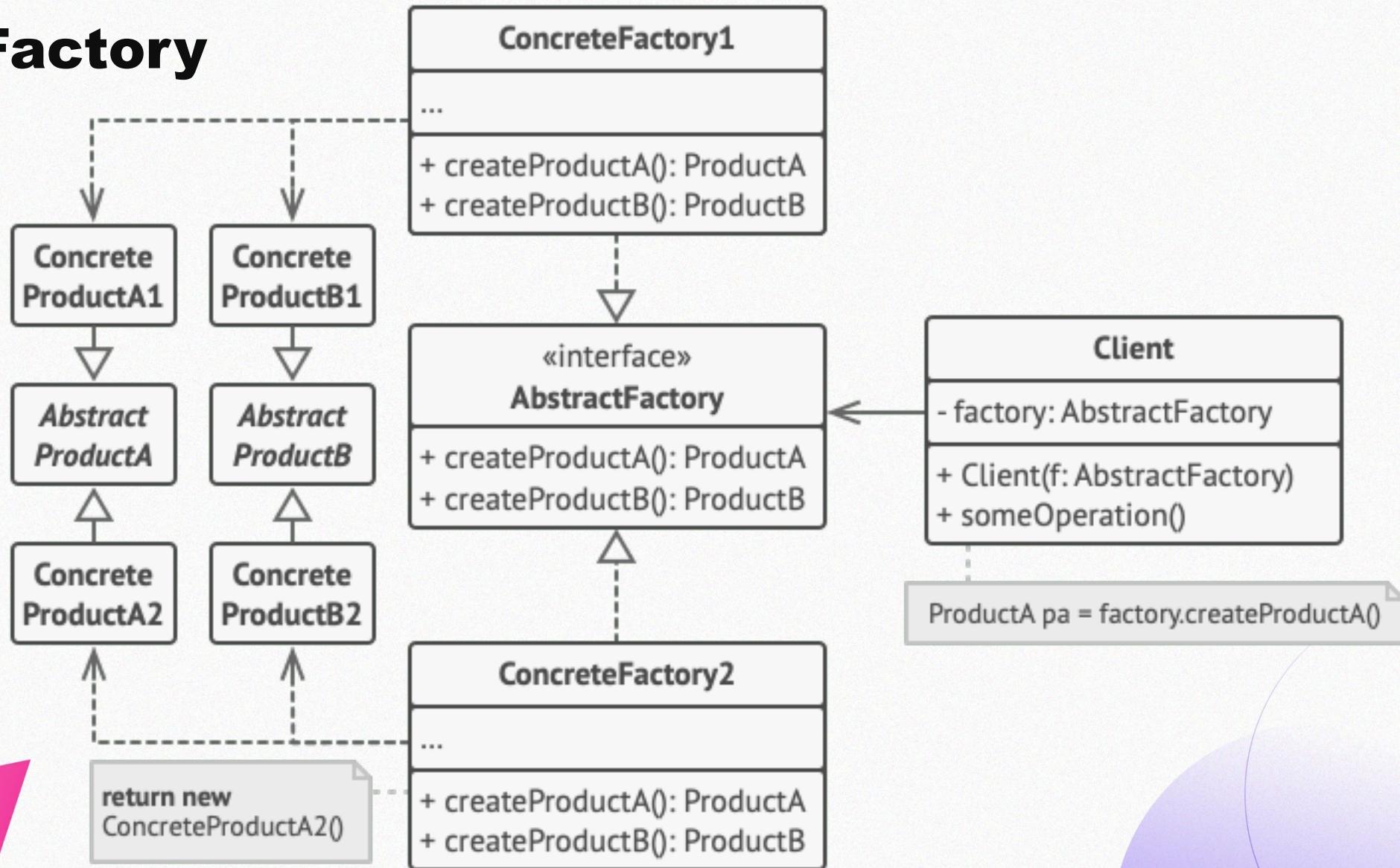
Protocolo: Los Message Brokers suelen usar protocolos estándar como AMQP, MQTT o STOMP.

Message Broker

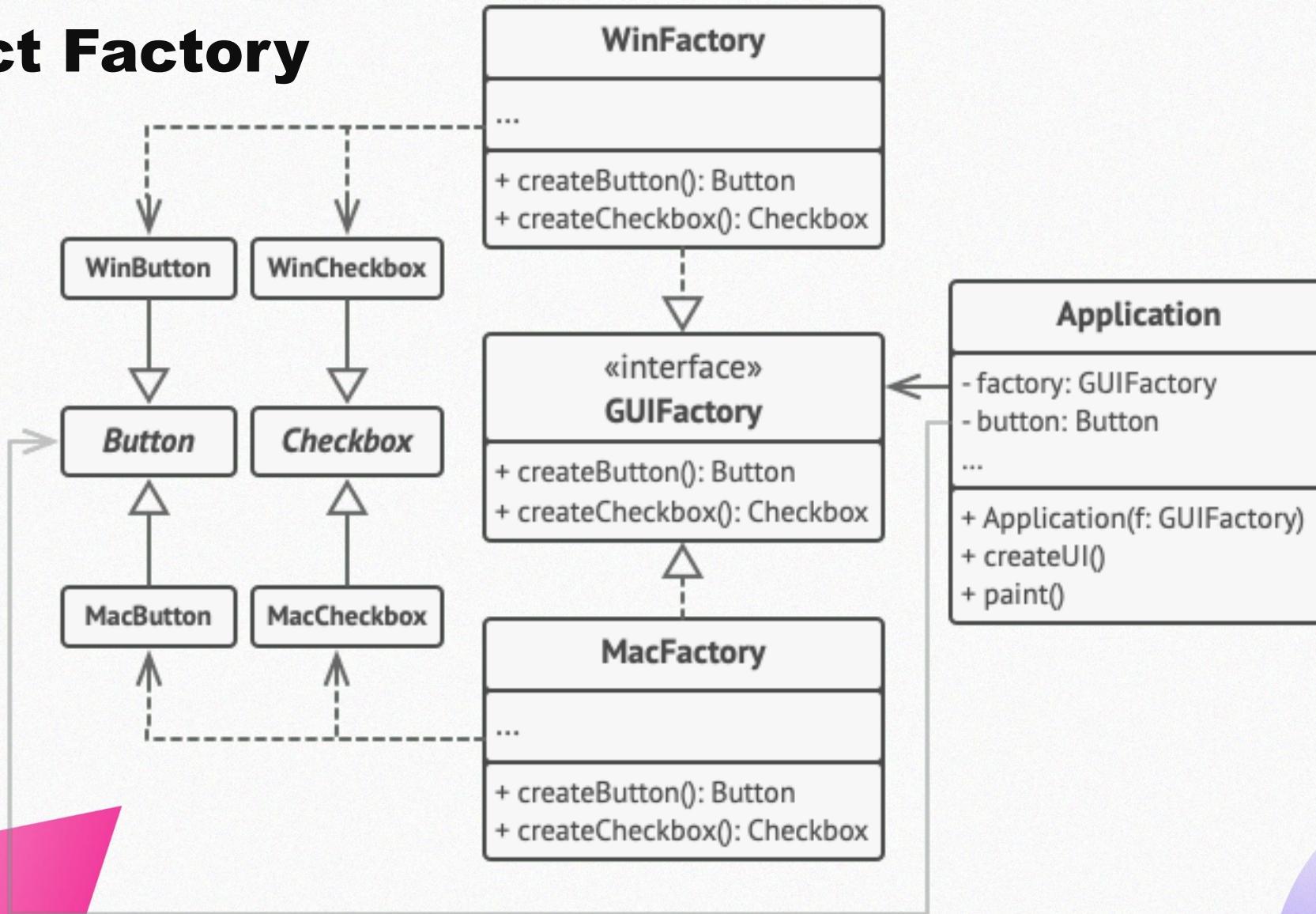
- **Point-to-Point** (P2P): Un mensaje va a un único consumidor (colas).
- **Publish/Subscribe** (Pub/Sub): Un mensaje se distribuye a múltiples consumidores suscritos.



Abstract Factory



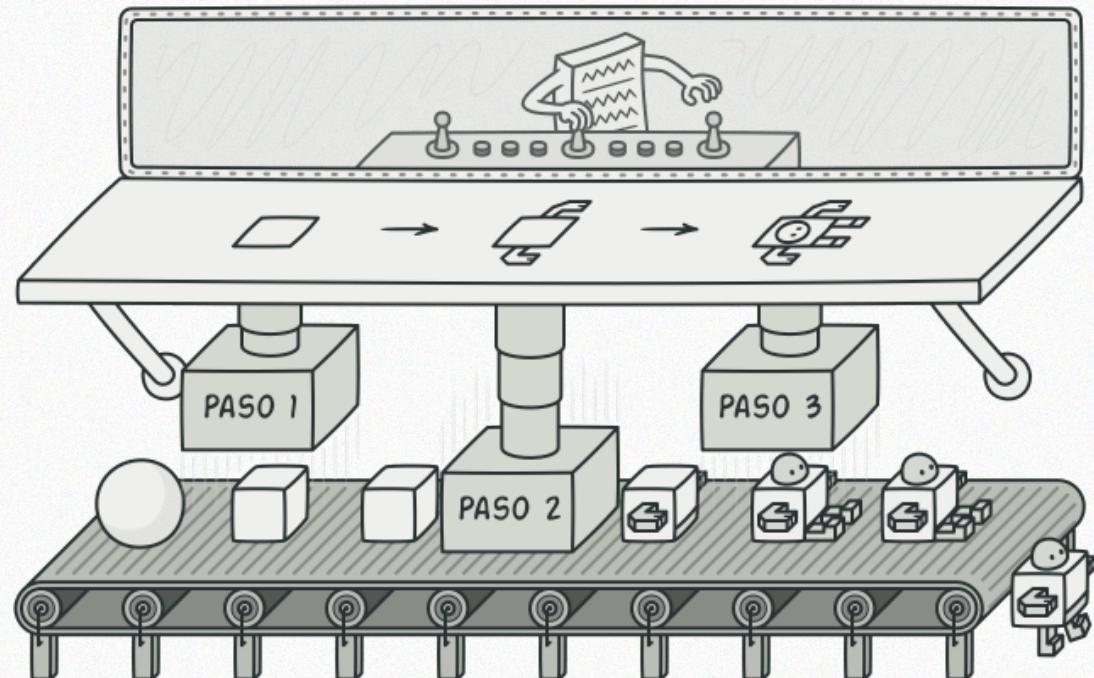
Abstract Factory



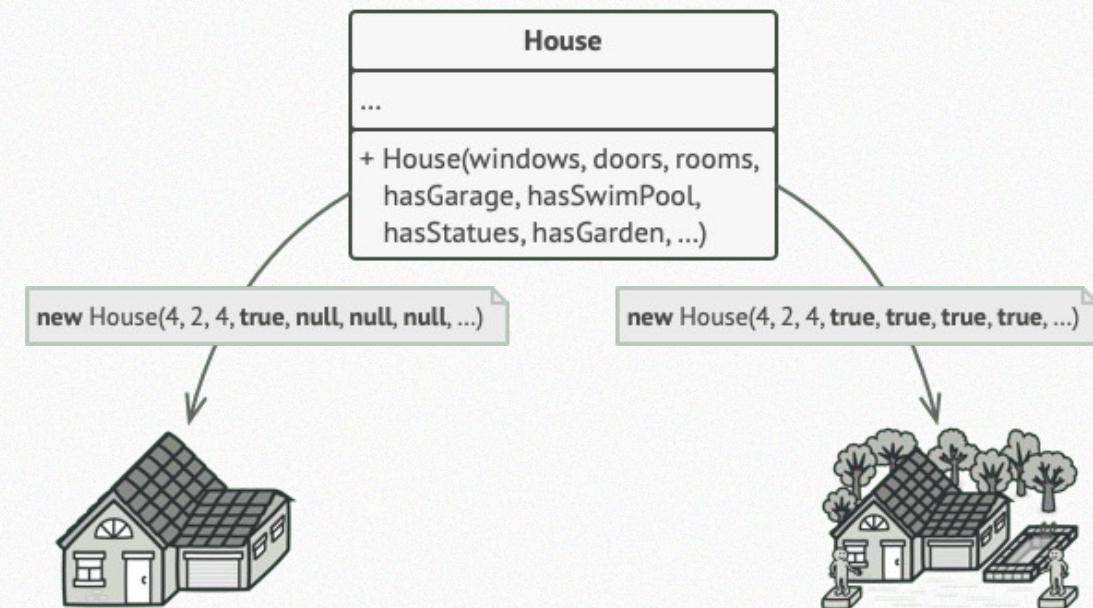
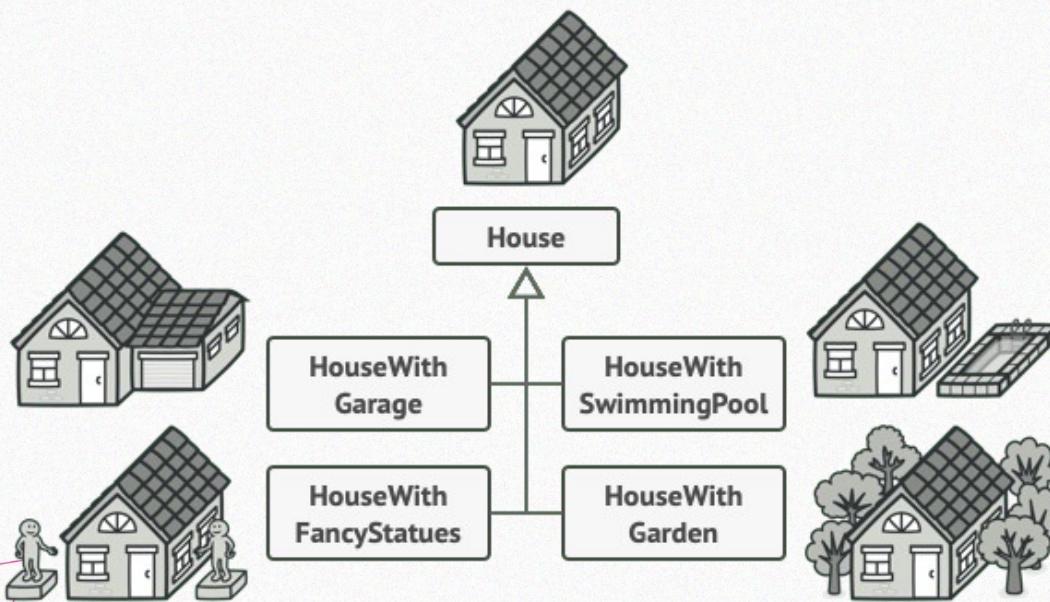
Builder

El patrón *Builder* separa la construcción de un objeto complejo de su representación, permitiendo crear diferentes representaciones con el mismo proceso de construcción.

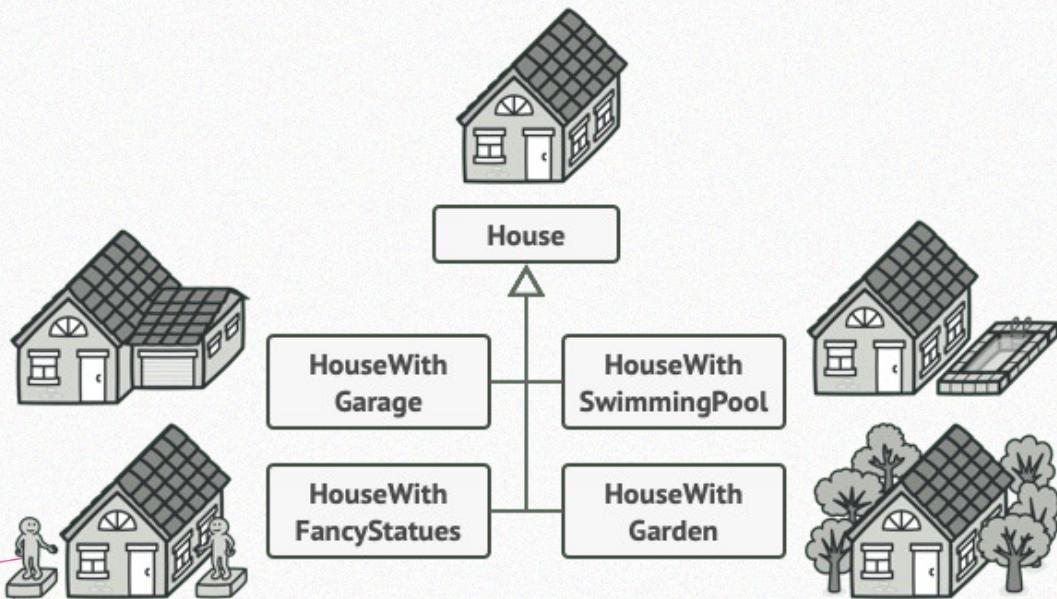
Este patrón es particularmente útil cuando los objetos tienen muchas configuraciones posibles.



Builder



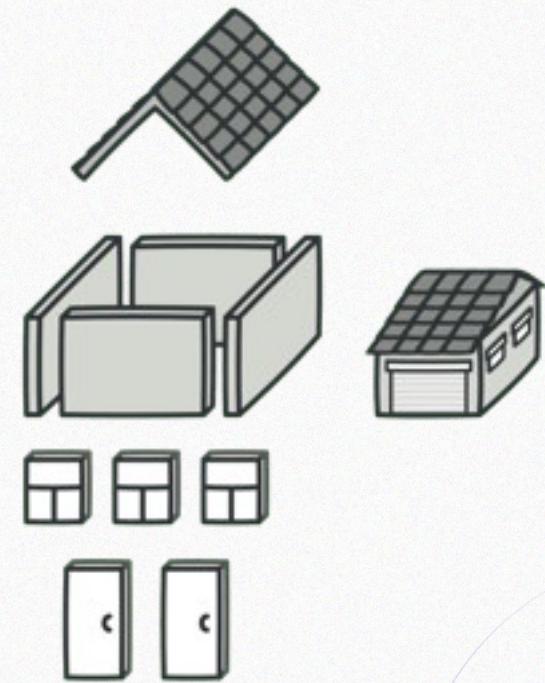
Builder



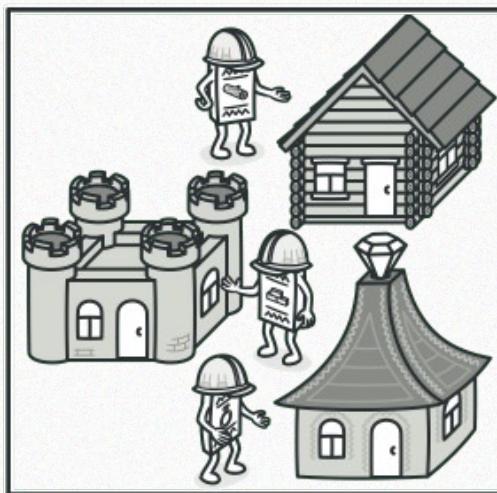
HouseBuilder

```

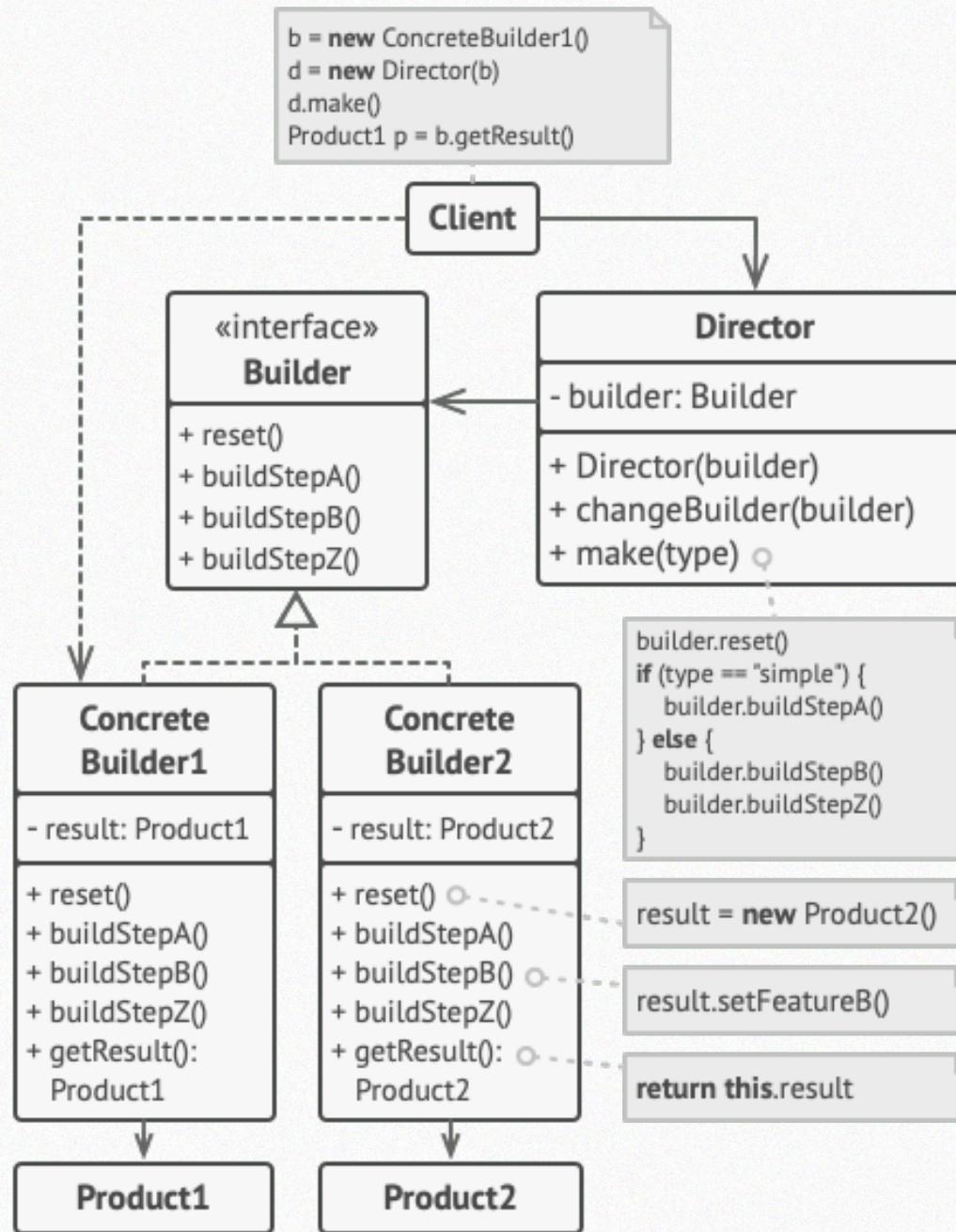
+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House
    
```



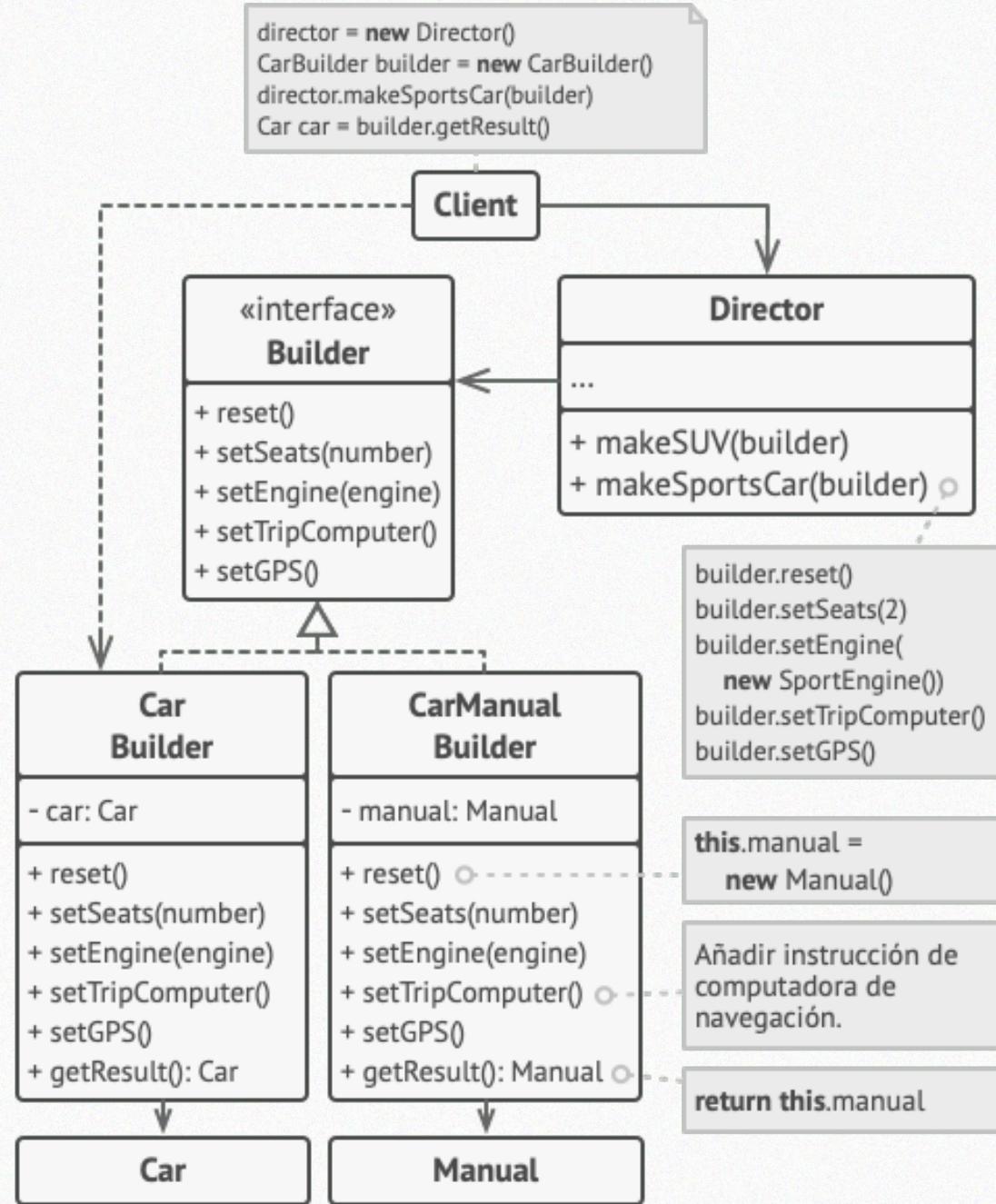
Builder



Builder

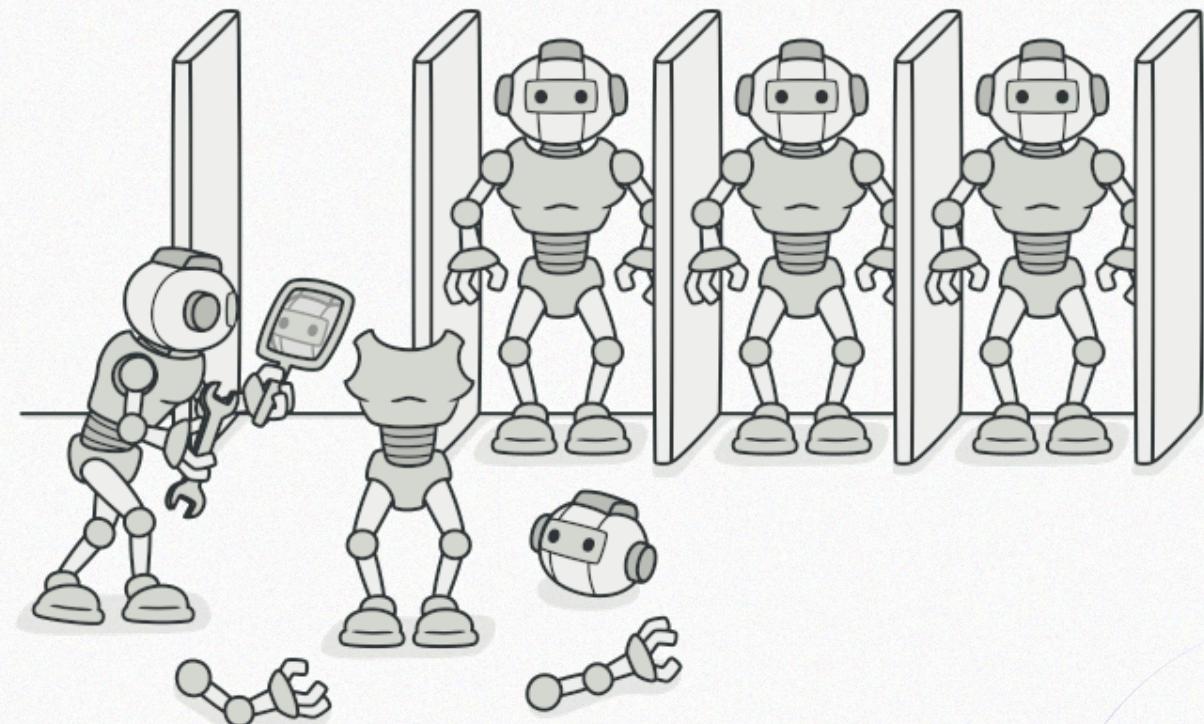


Builder



Prototype

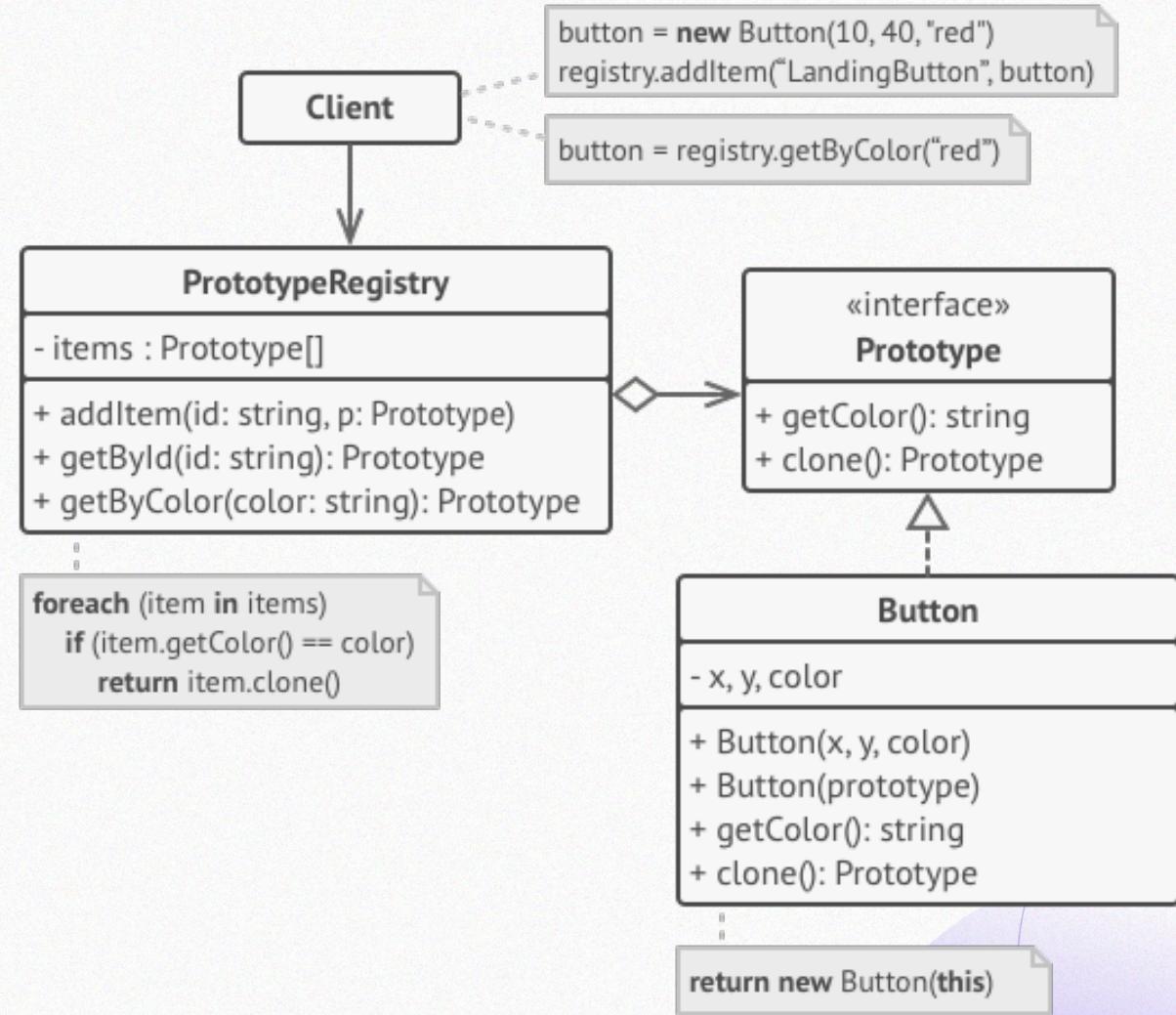
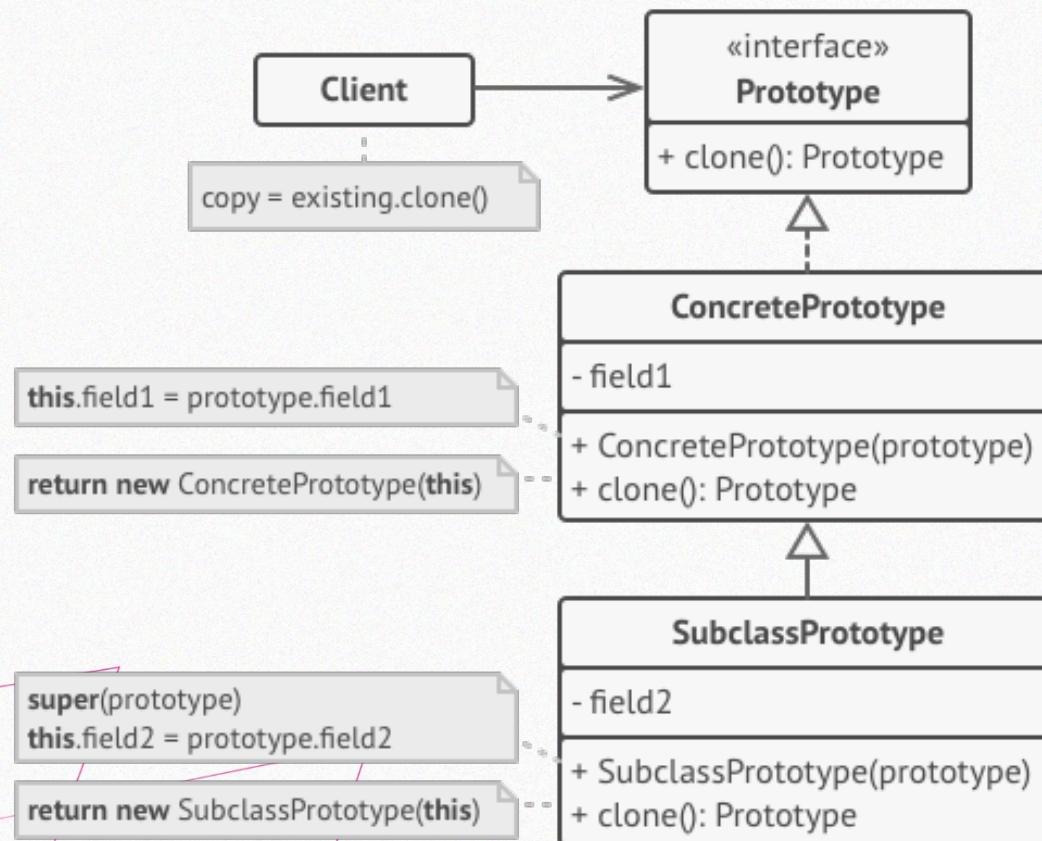
El patrón *Prototype* se utiliza cuando la creación de un objeto implica un coste significativo en tiempo o recursos. Este patrón permite copiar objetos existentes en lugar de crear nuevas instancias desde cero.



Prototype

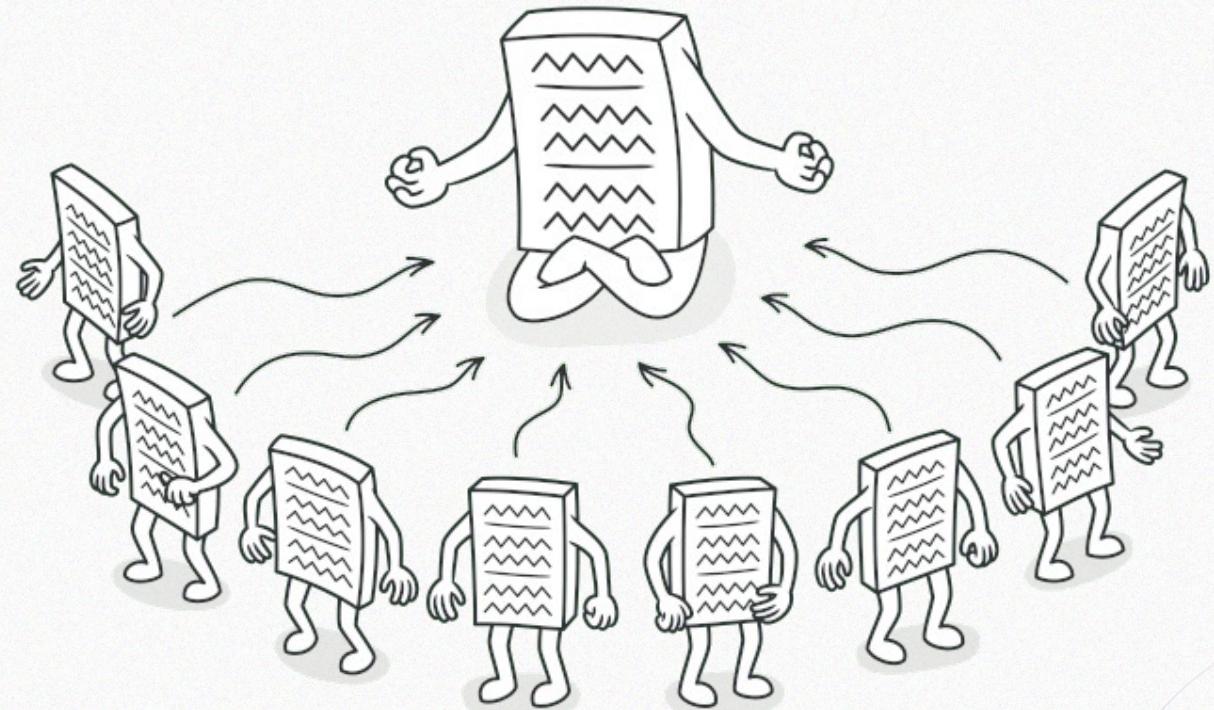


Prototype

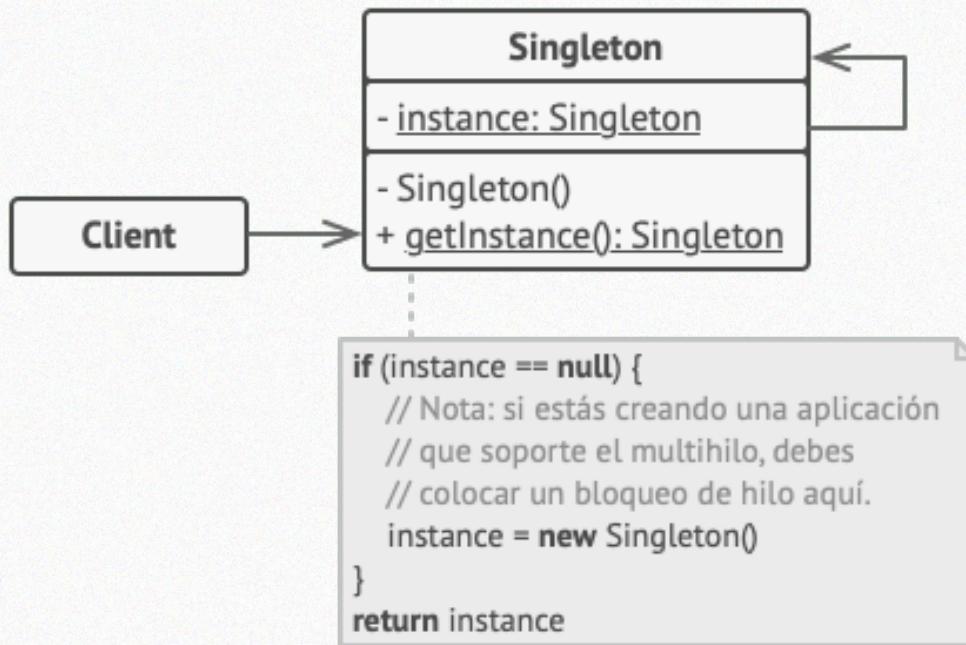


Singleton

El patrón *Singleton* garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esta. Es especialmente útil para objetos que representan recursos compartidos como conexiones a bases de datos o controladores de configuración.



Singleton



```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

The background features a dark gray gradient with several geometric shapes. In the upper left, there is a large pink triangle pointing downwards. Overlaid on it is a smaller purple triangle pointing upwards. A thin white line extends from the top-left corner towards the center. In the lower right, there is a large pink triangle pointing upwards. The text is centered within this triangular space.

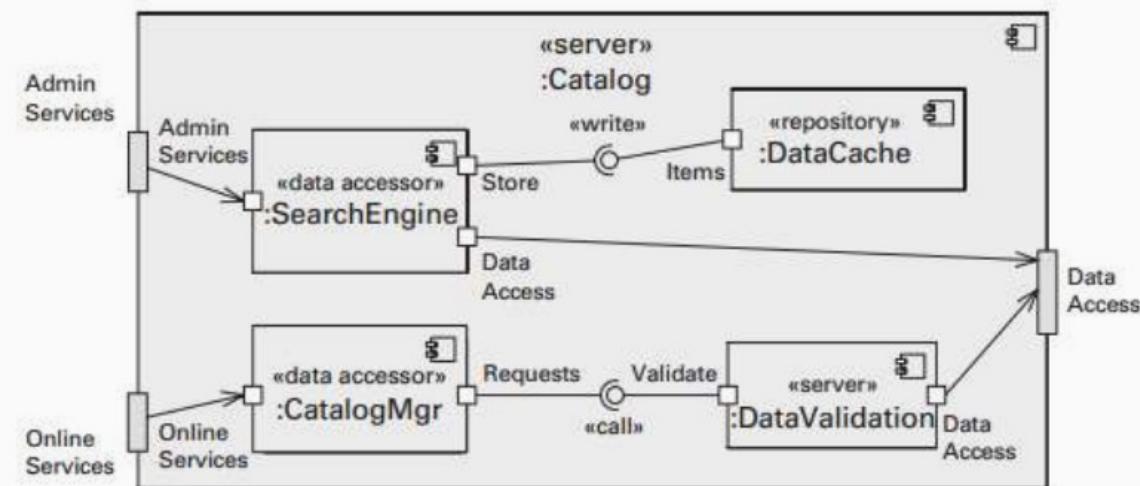
Componentes y Conectores

Vista de componentes y conectores

La vista de componentes y conectores muestra cómo interactúan las partes del sistema.

Componentes: unidades funcionales encapsuladas.

Conectores: mecanismos de interacción entre componentes.



Component

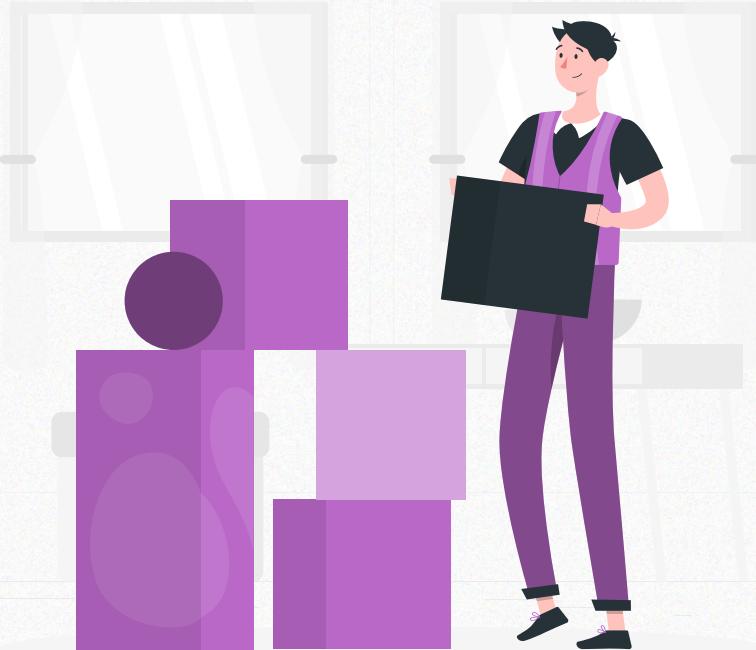
es

“Una unidad de composición de software con interfaces contractualmente definidas y explícitas, que permite su despliegue independiente y composición con otros componentes”.

Clemens Szyperski
(2002)

Características de los Componentes

- Encapsulación de funcionalidad.
- Interfaces explícitas para interoperabilidad.
- Reutilizables e independientes del contexto específico.



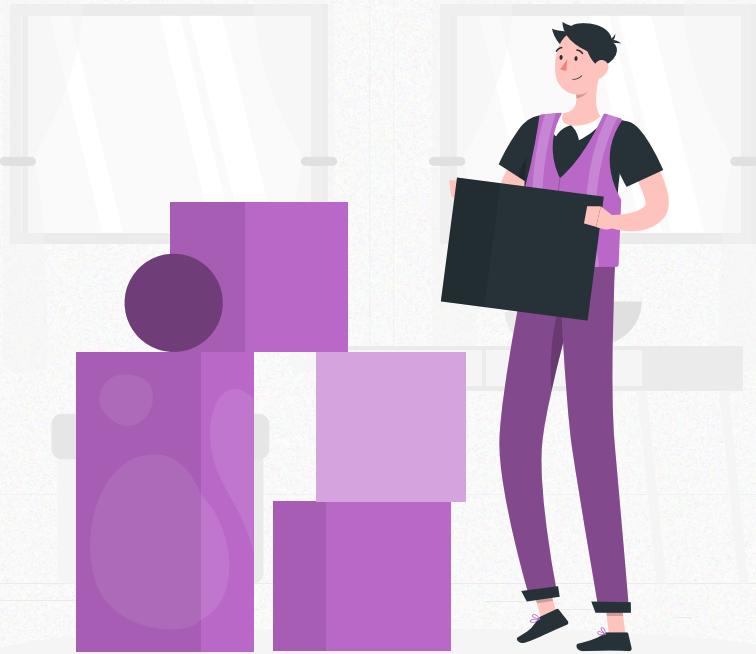
Conectores

“Un medio de comunicación entre componentes, que abstrae las interacciones desde llamadas locales hasta protocolos distribuidos”.

Bass, L., Clements, P., & Kazman, R.
(2013)

Tipos:

- **Síncrónicos:** RPC, HTTP.
- **Asíncrónicos:** EventBus, colas de mensajes.
- **De transmisión:** Flujos de datos continuos.

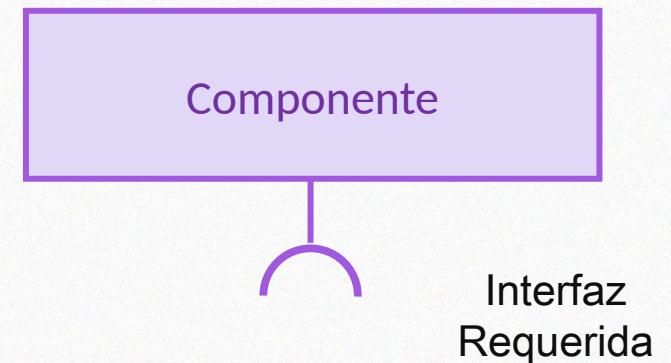
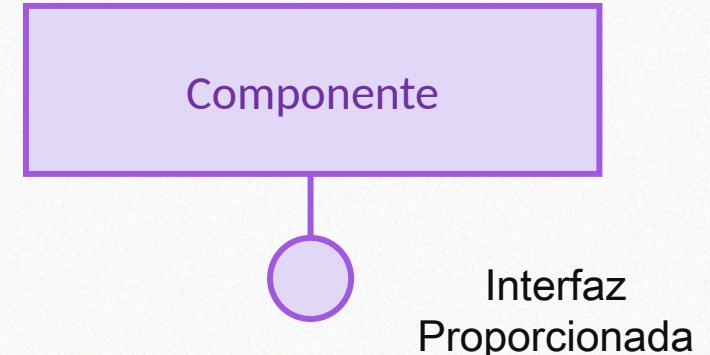


Interfaces

Las interfaces son elementos de componentes o clases que ofrecen funciones a otros componentes o clases. Definen operaciones que otros componentes o clases deben implementar.

Existen 2 tipos de interfaces:

- **Interfaces proporcionadas:** Servicios que el componente ofrece a sus clientes.
- **Interfaces requeridas:** Servicios que el componente necesita de otros componentes.

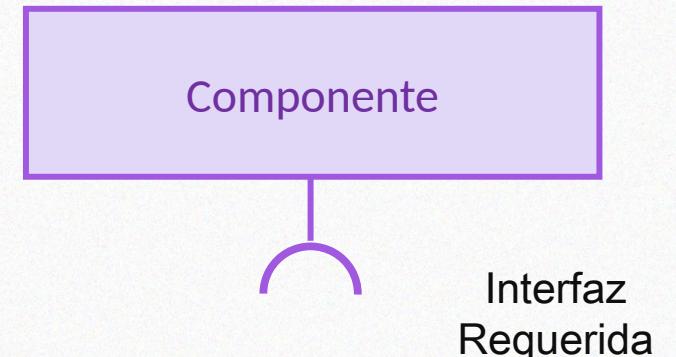
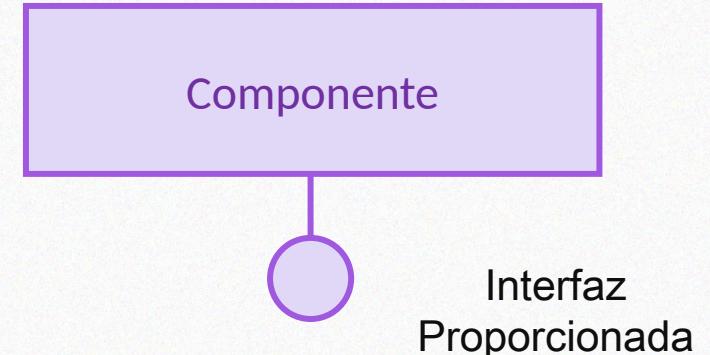


Interfaces

Las interfaces son elementos de componentes o clases que ofrecen funciones a otros componentes o clases. Definen operaciones que otros componentes o clases deben implementar.

Existen 2 tipos de interfaces:

- **Interfaces proporcionadas:** Servicios que el componente ofrece a sus clientes.
- **Interfaces requeridas:** Servicios que el componente necesita de otros componentes.



Interfaces



Interfaces

```
public interface BaseDeDatosIdentidad {  
    String obtenerIdentidad(String usuario);  
}
```

```
public interface Autenticable {  
    boolean autenticar(String usuario, String  
contraseña);  
}
```

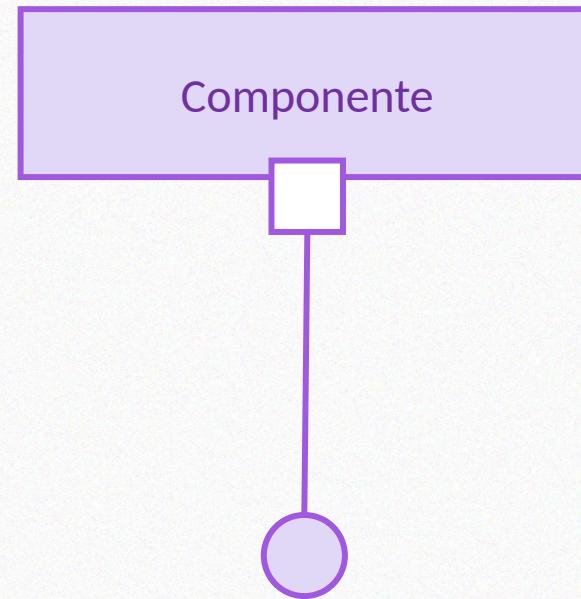
Interfaces

```
public class BaseDeDatosIAM implements BaseDeDatosIdentidad {  
    @Override  
    public String obtenerIdentidad(String usuario) {  
        return "juan".equals(usuario) ? "1234" : null;  
    }  
}  
  
public class ServicioDeAutenticacion implements Autenticable {  
    private final BaseDeDatosIdentidad baseDeDatos;  
  
    public ServicioDeAutenticacion(BaseDeDatosIdentidad baseDeDatos) {  
        this.baseDeDatos = baseDeDatos;  
    }  
  
    @Override  
    public boolean autenticar(String usuario, String contraseña) {  
        String identidad = baseDeDatos.obtenerIdentidad(usuario);  
        return identidad != null && identidad.equals(contraseña);  
    }  
}
```

Puertos

Los puertos son lugares donde dos cosas pueden conectarse. Los símbolos para los puertos son pequeños círculos o rectángulos en el límite de un componente.

Permiten que los componentes se comuniquen entre sí. Por ejemplo, el componente de autenticación podría tener un puerto para recibir información de inicio de sesión del usuario.



Interfaces

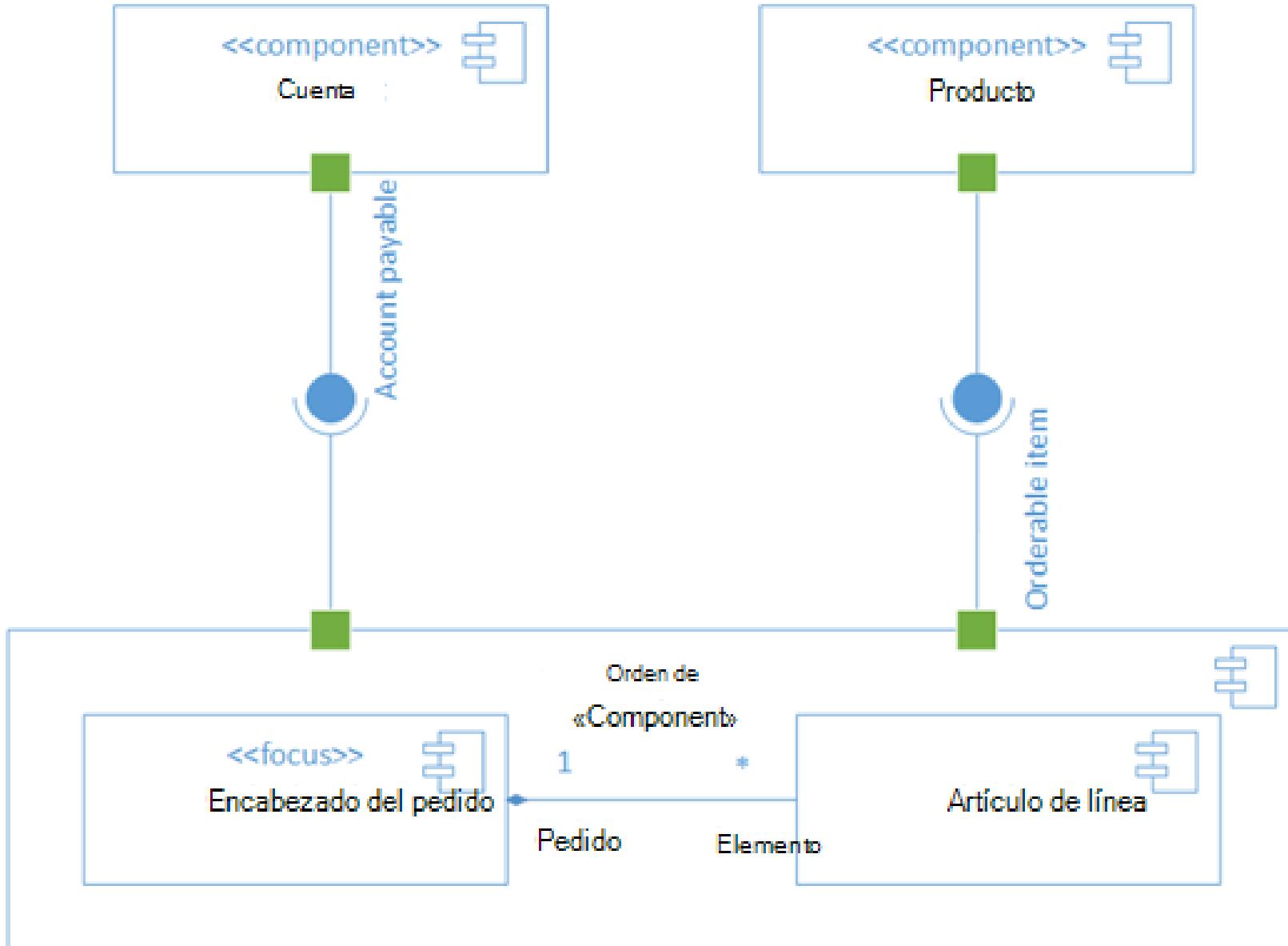


Puerto

```
public class ServicioDeAutenticacion {  
  
    private final PuertoAutenticacion puerto;  
  
    public ServicioDeAutenticacion(BaseDeDatosIdentidad baseDeDatos) {  
        this.puerto = new PuertoAutenticacion(baseDeDatos);  
    }  
  
    public Autenticable obtenerPuerto() {  
        return puerto;  
    }  
    private class PuertoAutenticacion implements Autenticable {  
        private final BaseDeDatosIdentidad baseDeDatos;  
  
        public PuertoAutenticacion(BaseDeDatosIdentidad baseDeDatos) {  
            this.baseDeDatos = baseDeDatos;  
        }  
  
        @Override  
        public boolean autenticar(String usuario, String contraseña) {  
            String identidad = baseDeDatos.obtenerIdentidad(usuario);  
            return identidad != null && identidad.equals(contraseña);  
        }  
    }  
}
```

Dependencias



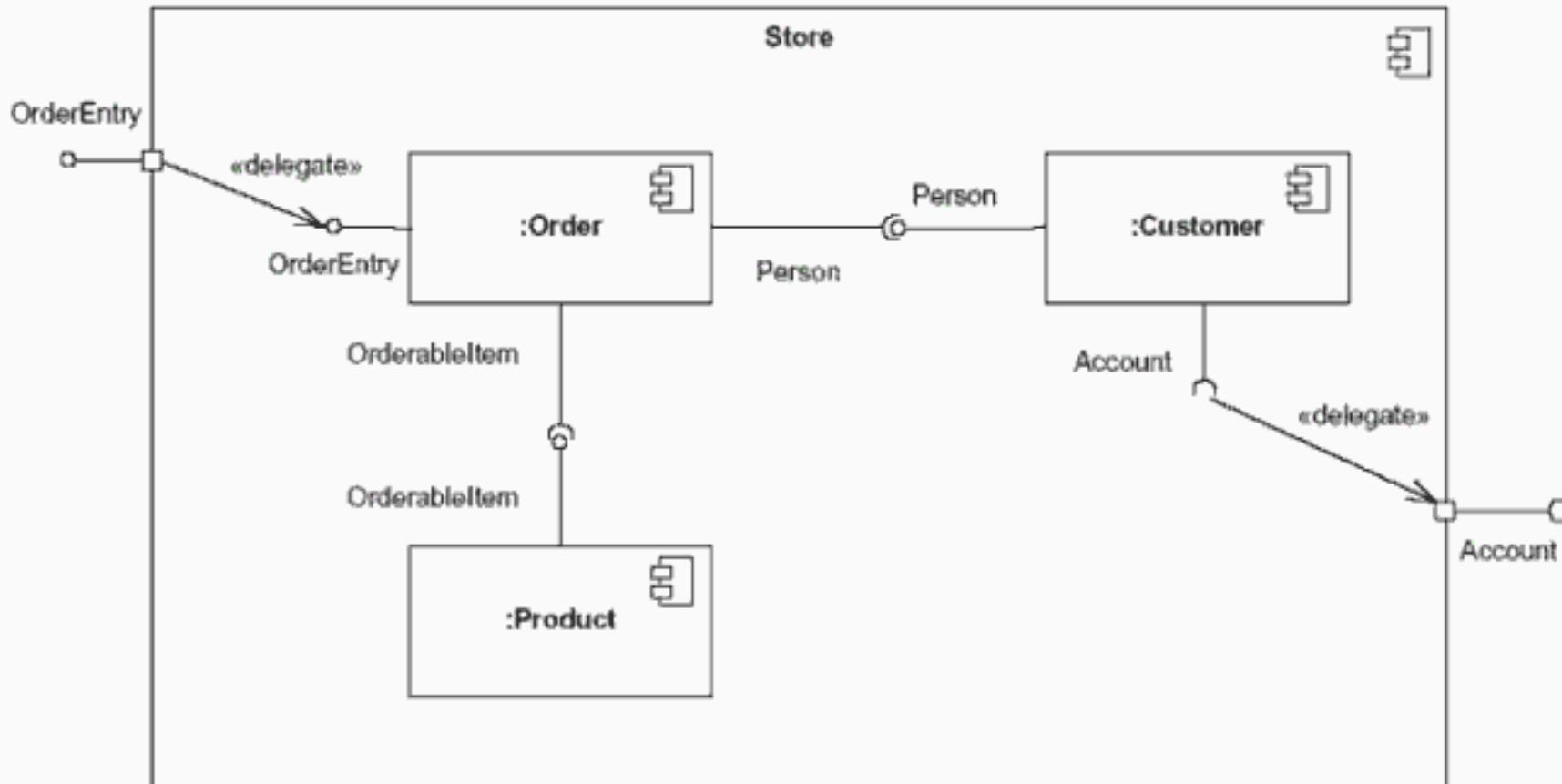


Conectores

Tipos de Conectores:

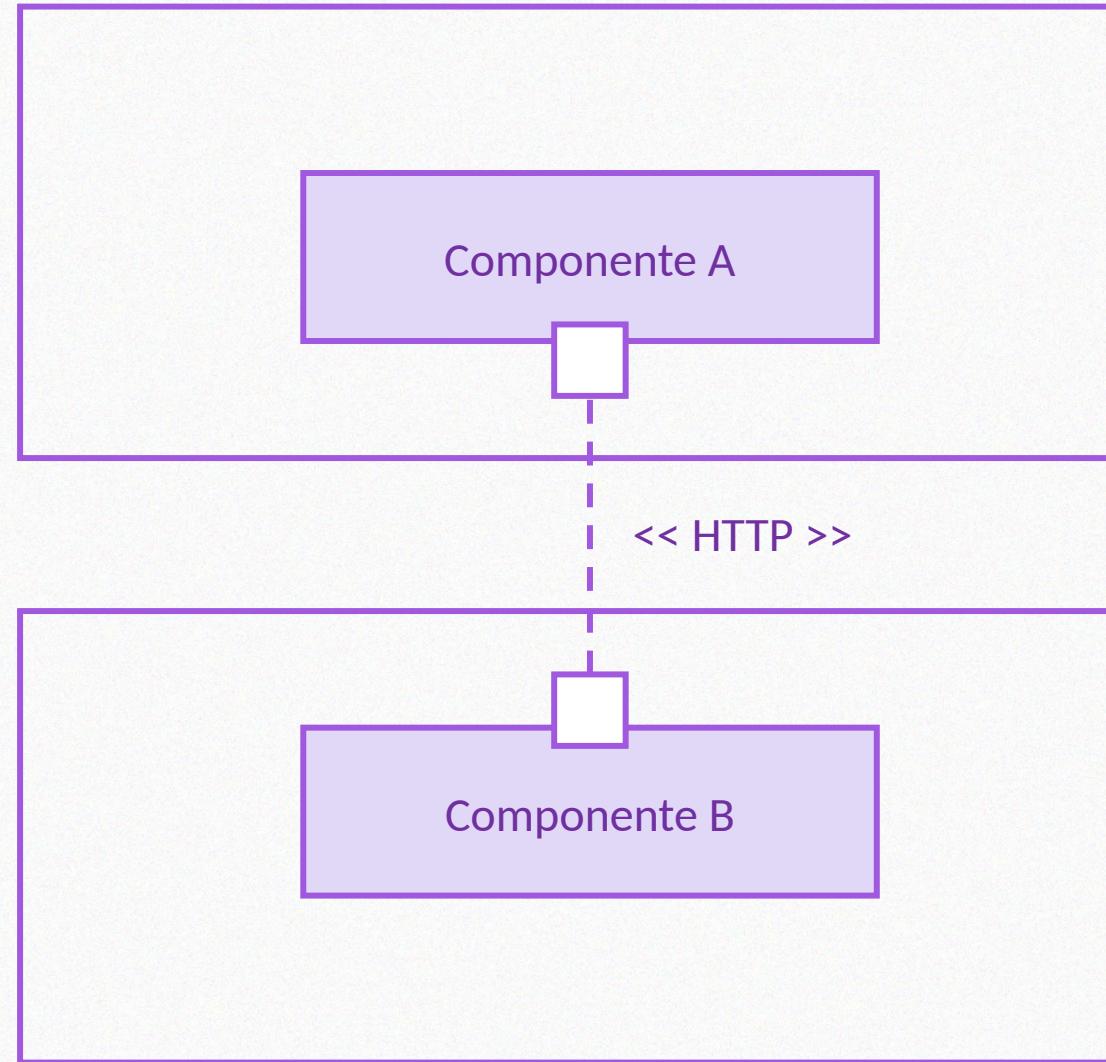
- **Sincrónicos:** RPC, llamadas directas (gRPC, HTTP en modo sincrónico).
- **Asincrónicos:** Mensajería basada en eventos (RabbitMQ, Kafka).
- **De transmisión:** Streaming (WebSockets, gRPC bidireccional).
- **Intermediarios:** Proxies, Gateways, Middleware.

Conectores



Conectores

Client-server - permiten que un conjunto de clientes concurrentes recuperen datos sincrónicamente a través de requerimientos de servicios



Conectores

```
import spark.Spark;

public class ComponenteB {
    public static void main(String[] args) {
        // Configurar el puerto del servidor
        Spark.port(8080);

        // Endpoint HTTP que expone un servicio
        Spark.get("/data", (req, res) -> {
            res.type("application/json");
            return "{ \"message\": \"Hola desde Componente B\" }";
        });

        System.out.println("Componente B listo en http://localhost:8080/data");
    }
}
```

Conectores

```
import spark.Spark;

public class ComponenteB {
    public static void main(String[] args) {
        // Configurar el puerto del servidor
        Spark.port(8080);

        // Endpoint HTTP que expone un servicio
        Spark.get("/data", (req, res) -> {
            res.type("application/json");
            return "{ \"message\": \"Hola desde Componente B\" }";
        });

        System.out.println("Componente B listo en http://localhost:8080/data");
    }
}
```

Conectores

```
import spark.Spark;
public class PuertoServidor {
    public void start() {
        Spark.port(8080);

        Spark.get("/data", (req, res) -> {
            res.type("application/json");
            return "{ \"message\": \"Hola desde Componente B\" }";
        });
        System.out.println("PuertoServidor listo en http://localhost:8080/data");
    }
}

public class ComponenteB {
    public static void main(String[] args) {
        PuertoServidor puertoServidor = new PuertoServidor();
        puertoServidor.start(); // Iniciar el puerto proporcionado
    }
}
```

Conectores

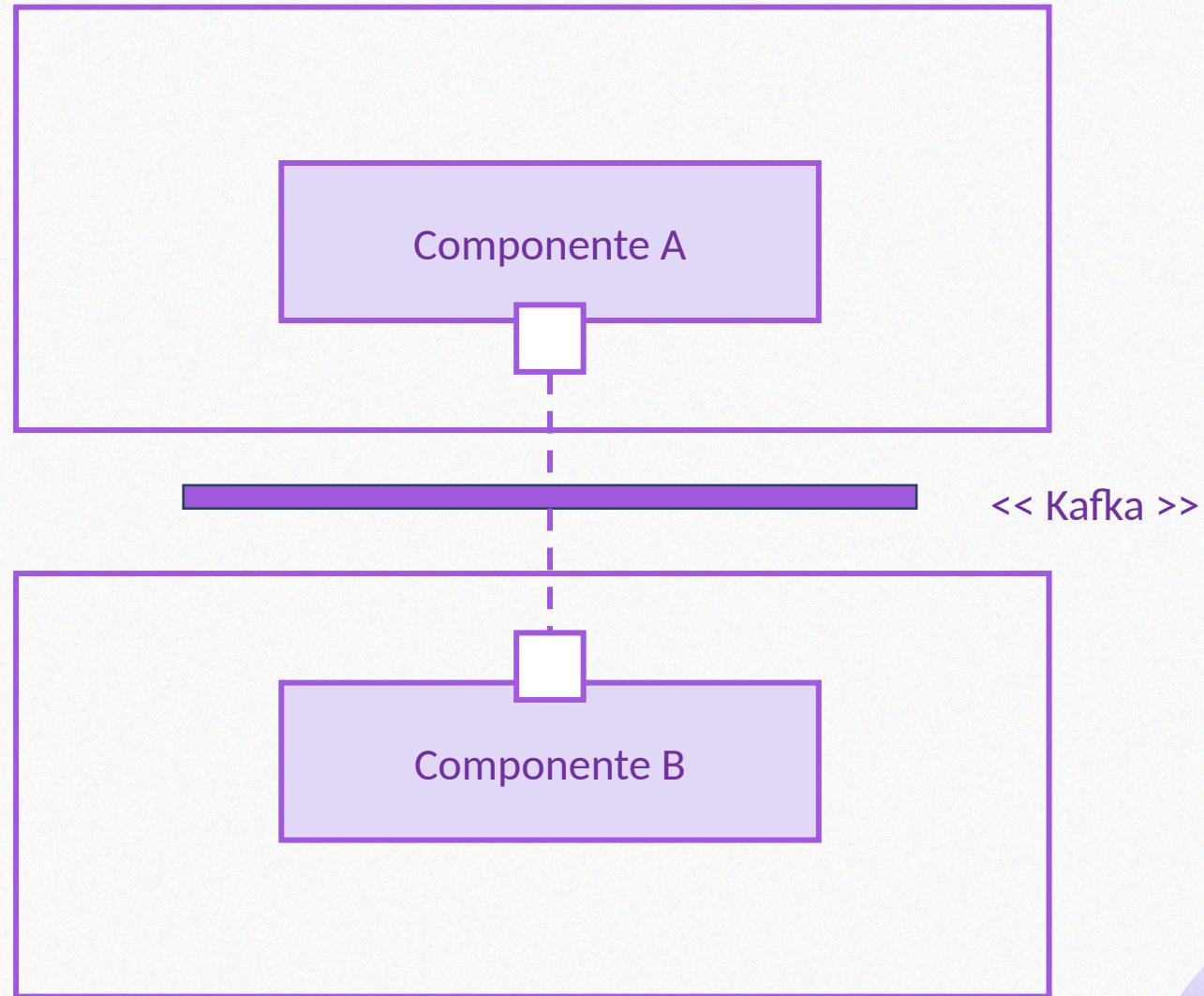
```
public class PuertoCliente {  
    private final String baseUrl;  
  
    public PuertoCliente(String baseUrl) {  
        this.baseUrl = baseUrl;  
    }  
  
    public String obtenerDatos() {  
        // Realizar solicitud HTTP GET al servidor  
        HttpResponse<JsonNode> respuesta = Unirest.get(baseUrl + "/data").asJson();  
  
        // Retornar el mensaje recibido  
        if (respuesta.getStatus() == 200) {  
            return respuesta.getBody().getObject().getString("message");  
        } else {  
            throw new RuntimeException("Error al conectar con el servidor: " +  
                respuesta.getStatus());  
        }  
    }  
}
```

Conectores

```
public class ComponenteA {  
    public static void main(String[] args) {  
        // Crear el puerto requerido  
        PuertoCliente puertoCliente = new PuertoCliente("http://localhost:8080");  
  
        // Usar el puerto para interactuar con el servidor  
        String respuesta = puertoCliente.obtenerDatos();  
        System.out.println("Respuesta del Componente B: " + respuesta);  
    }  
}
```

Conectores

Asincrono - Soporta envío de eventos y notificaciones asincrónicas.



Conectores

Transmision -
Soporta el flujo
continuo de
informacion

